

COVID Open Data

Rapport technique

Kawtar HAMDI
Lisa ESTEBE
Salomé CHEVAILLER
Stéphane SUN

Sommaire

Sommaire	1
Introduction	2
I) Présentation générale de l'application	3
II) Modèle de données	4
III) Structure de l'application	5
Model	5
Classes-entités	5
Repositories	5
ScheduledTaskTest	7
downloadFile()	7
verificateur()	8
Méthodes link	8
Méthodes save	9
Controllers	10
ContinentController	10
ContinentApiController	11
CountryApiController	11
InfoDailyCountryApiController	12
WorldApiController	12
Views	13
maps.js	13
graphs.js	16
IV) Pour aller plus loin	18
V) Gestion de projet	19
Annexes	20
Diagramme UML initial	20
Diagramme UML final	21
Maquette	22

Introduction

L'objectif de notre projet intitulé “Covid Open Data” était de pouvoir automatiser l'importation d'un sous-ensemble de jeux de données “ouverts” sur l'épidémie de COVID-19 dans une base de données relationnelle, pour pouvoir par la suite développer une application Web permettant de faire des requêtes et de visualiser les résultats sur ce sous-ensemble. Pour cela, nous avons dû sélectionner les jeux de données qu'on estimait être pertinents, définir un modèle conceptuel de données correspondant et l'implémenter dans une base de données relationnelle. Ensuite, nous avons importé le sous-ensemble du jeu de données dans le SGBD et nous avons développé une application Web permettant de requêter ce jeu de données et de visualiser les résultats sous plusieurs types de graphiques.

Dans ce rapport, nous ferons tout d'abord une présentation générale de l'application, puis nous commenterons notre modèle de données. Nous détaillerons ensuite toute la structure de l'application et aborderons les problèmes non résolus, les fonctionnalités restant à développer et les possibles évolutions. Pour finir, nous présenterons les autres aspects techniques utilisés tout au long du projet.

I) Présentation générale de l'application

Notre application web permet d'obtenir des visualisations graphiques à partir de données journalières.

La page principale propose une carte du monde avec un tableau renseignant les données du jour. Deux graphiques montrant l'évolution des données sont également visibles.

Il est possible de choisir quelle région du monde nous intéresse en sélectionnant le continent souhaité. On obtient ainsi la carte du continent choisi ainsi que les données et graphiques correspondants.

Il est également possible de choisir quelle information afficher lorsque l'on passe la souris sur les pays de la carte, en sélectionnant celle-ci dans le menu proposé.

Lorsqu'on veut afficher les données d'un pays, il suffit de cliquer dessus et celles-ci apparaissent dans le tableau, ainsi que sur les graphiques.

Vous trouverez les instructions d'utilisation dans le Readme.



II) Modèle de données

Dans un premier temps, nous avons fait énormément de recherches afin de trouver des données ouvertes pertinentes sur le COVID-19 et régulièrement mises à jour.

Nous avons finalement retenu deux sources différentes. Une première, Our World in Data, permettant de récupérer des données sur les pays du monde entier ([Données coronavirus des pays du monde](#)), et la seconde, data.gouv.fr, récupérant les données des différents départements de la France ([Données coronavirus des départements français](#)). Chacun comportant un grand nombre de données, nous avons dû trier et récupérer les données les plus intéressantes pour notre application.

Notre idée de départ était de faire plusieurs visualisations graphiques : une pour le monde entier, une pour l'Europe et une pour la France. A partir de cela nous avons donc établi un premier diagramme UML que vous trouverez en annexe (page 20).

Mais en commençant à coder, nous nous sommes rendus compte que notre idée de départ était un peu trop ambitieuse et nous avons décidé de nous concentrer uniquement sur les données du monde entier et non plus en faisant une focalisation sur les données de la France. Seule la source Our World in Data a donc servi à notre projet. Nous avons alors modifié notre diagramme UML que vous trouverez en annexe (page 21).

Voici quelques explications sur celui-ci :

Il est composé d'une classe-entité Continent ayant pour attribut son nom et différentes méthodes permettant d'obtenir toutes les informations journalières d'un continent (getInfosContinentByName()) et toutes les informations journalières du monde entier à une date donnée (getInfosWorld()). Cette classe-entité est elle-même composée de la classe-entité Country.

Cette dernière comporte plusieurs attributs, notamment son code et son nom. Elle a également différentes méthodes permettant d'obtenir une liste des pays par continent (getCountriesByContinent()) et toutes les informations journalières d'un pays à une date donnée (getCountryByName()). Cette classe-entité est associée à la classe-entité InfoDailyCountry.

La classe-entité InfoDailyCountry comporte elle aussi plusieurs attributs dont le nom est assez explicite. Elle a également des méthodes permettant d'obtenir le code du pays correspondant aux infos journalière à une date donnée (getIdInfoCountryByCountryInformedCodeCountryAndDate()), toutes les données journalières pour chaque pays (getAllDailyStatsCountryByName()) et le nombre de nouveaux cas à une date précise (getNewCases()).

Toutes les valeurs des attributs de nos classes-entités sont directement récupérées dans le fichier Our World in Data.

III) Structure de l'application

Notre application suit le principe de structuration MVC (Model-View-Controller). Nous allons donc vous présenter ces trois aspects.

Model

Classes-entités

Le “Model” est composé des objets qui sont dans la base de données. Il s’agit donc des classes-entités JPA (Java Persistence API) et des classes DAO (Data Access Object) qui les manipulent.

Dans notre application, les classes-entités sont celles que l’on retrouve dans notre diagramme UML. Nous avons également établi le Mapping des relations UML en relationnel. Voici un exemple avec la classe-entité Continent:

```
1 package covid.entity;
2 import java.util.LinkedList;
3 import java.util.List;
4 import javax.persistence.*;
5 import lombok.*;
6
7 // Un exemple d'entité
8 // On utilise Lombok pour auto-générer getter / setter / toString...
9 // cf. https://examples.javacodegeeks.com/spring-boot-with-lombok/
10 @Getter @Setter @NoArgsConstructor @RequiredArgsConstructor @ToString
11 @Entity // Une entité JPA
12 public class Continent {
13
14     // Attributs
15     @Column (unique=true)
16     @NonNull
17     @Id
18     private String nameContinent;
19
20     // Relations
21     @OneToMany(fetch=FetchType.EAGER, mappedBy = "continent", cascade = CascadeType.ALL)
22     List<Country> countries = new LinkedList<Country>();
23
24     // Ajout de Country
25     public void addCountry(Country country) {
26         this.countries.add(country);
27         country.setContinent(this);
28     }
29 }
```

Repositories

Nous avons également créé des classes DAO (Repository) pour chaque classe-entité. Ces classes sont auto-implémentées par Spring à partir de la déclaration d'une interface.

Pour le DAO de la classe-entité Country, nous avons ajouté six requêtes SQL. La première permettant d'obtenir une liste des pays pour chaque continent. La deuxième permet quant à elle d'obtenir toutes les informations d'un pays à la date souhaitée. La troisième permet de récupérer les informations d'un continent en fonction de son nom à une date donnée. Notre quatrième requête SQL permet d'obtenir toutes les informations du monde entier à une date souhaitée. La cinquième permet d'obtenir le nombre de morts total et la dernière le nombre de cas total. Voici les deux premières :

```

14  @Repository
15  public interface CountryRepository extends JpaRepository<Country, String> {
16
17      // Requête permettant de récupérer la liste des Country en fonction de leur Continent
18      @Query(value = "SELECT name_Country AS name, total_Cases AS cases "
19             + "FROM Country "
20             + "WHERE Continent_Name_Continent LIKE %:nameContinent% "
21             , nativeQuery = true)
22      List<Object> getCountriesByContinent(@Param("nameContinent") String nameContinent);
23
24      // Requête permettant de récupérer les infos d'un Country en fonction de la date
25      @Query(value = "SELECT c.code_Country AS code, c.name_Country AS name, c.total_Cases AS tcases, c.total_Deaths AS tdeaths, "
26             + "c.icu_Patients AS icu, c.hosp_Patients AS hosp, c.total_Tests AS ttést, c.total_Vaccinations AS tvaccinations, "
27             + "c.fully_Vaccinated AS fvaccinated, c.stringency_Index AS si, c.population AS pop, c.gdp AS gdp, "
28             + "i.new_Cases AS ncases, i.new_Deaths AS ndeaths, i.new_Vaccinations AS nvaccinations "
29             + "FROM Country c "
30             + "INNER JOIN Info_Daily_Country i "
31             + "ON c.code_Country=i.country_informed_code_country "
32             + "WHERE c.name_Country = :nameCountry "
33             + "AND i.date = :date "
34             , nativeQuery = true)
35      InfoCountry getInfoCountryByName(@Param("nameCountry") String nameCountry, @Param("date") LocalDate date);
36

```

Pour le DAO de la classe-entité InfoDailyCountry, nous avons ajouté cinq requêtes SQL. La première permet d'obtenir le code du pays correspondant à chaque InfoDailyCountry à une certaine date. La deuxième requête permet d'obtenir le nombre de nouveaux cas par pays à une date donnée. La troisième permet d'obtenir le nombre de nouvelles morts par pays. La quatrième permet d'obtenir toutes les données journalières par pays. La cinquième permet quant à elle d'obtenir toutes les données journalières par continent. La sixième permet de récupérer les données journalières du monde entier. Et les trois dernières permettent d'obtenir les nouvelles morts et nouveaux cas par pays, continent et dans le monde. Voici les deux premières :

```

10  public interface InfoDailyCountryRepository extends JpaRepository<InfoDailyCountry, Integer> {
11
12      // Requête permettant de récupérer l'ID à partir de la date et du codeCountry
13      @Query(value = "SELECT Id_Info_Country AS id FROM Info_Daily_Country "
14             + "WHERE Country_Informed_Code_Country = :codeCountry "
15             + "AND Date = :date ",
16             nativeQuery = true)
17      Integer getIdInfoCountryByCountryInformedCodeCountryAndDate(@Param("codeCountry") String codeCountry, @Param("date") LocalDate date);
18
19      // Requête permettant de récupérer le nombre de nouveaux cas en fonction de la date
20      @Query(value = "SELECT Country.name_Country AS name, Info_Daily_Country.new_Cases AS ncases "
21             + "FROM Info_Daily_Country "
22             + "INNER JOIN Country "
23             + "ON Info_Daily_Country.Country_Informed_Code_Country=Country.Code_Country "
24             + "WHERE Info_Daily_Country.date = :date ",
25             nativeQuery = true)
26      List<Object> getNewCases(@Param("date") LocalDate date);
27

```

Avant de passer à la partie Controller de notre application, il est nécessaire de présenter notre classe ScheduledTaskTest.

ScheduledTaskTest

downloadFile()

Etant donné que nous récupérons des données en ligne, il a été nécessaire de créer un timer s'exécutant dans la seconde après que le fichier ait été “run” (initialDelay = 1000) et qui exécute la tâche spécifiée toutes les 24h (fixedRate = 86400000). Ce timer exécute donc la méthode downloadFile() qui permet de télécharger et stocker les données du fichier Our World in Data. Cette méthode est séparée en plusieurs parties. Tout d'abord, on a lié notre instance et l'URL du fichier, créé un chemin “dataOWD.csv” et indiqué qu'il fallait stocker les données téléchargées dans ce chemin.

```
67
68 // Téléchargement fichier OWD
69 @Scheduled(initialDelay = 1000, fixedRate = 86400000)
70 public void downloadFile() {
71     // Code
72     try {
73         // Téléchargement et stockage du fichier OWD
74         // On crée un lien direct entre l'url et notre instance
75         ReadableByteChannel readChannel = Channels.newChannel(new URL("https://covid.ourworldindata.org/data/owid-covid-data.csv").openStream());
76         // On crée un chemin nommé "dataOWD.csv"
77         FileOutputStream fileOS = new FileOutputStream("dataOWD.csv");
78         // On stocke les données téléchargées dans le chemin
79         FileChannel writeChannel = fileOS.getChannel();
80         writeChannel
81             .transferFrom(readChannel, 0, Long.MAX_VALUE);
```

La deuxième partie concerne le traitement du fichier téléchargé. Pour cela, on vient transformer le chemin en fichier csv et créer une instance de lecture qui permet de lire le csv. Et on indique enfin que nos données sont séparées par une virgule.

```
82
83     // Traitement du fichier OWD
84     // On transforme le chemin en fichier csv
85     File fileOWD = new File("dataOWD.csv");
86     // On crée une instance de lecture pour lire le csv
87     FileReader frOWD = new FileReader(fileOWD);
88     // On définit le séparateur (pour le csv : ",")
89     CSVReader csvrOWD = new CSVReader(frOWD, ',');
90
```

La troisième partie s'agit de la lecture du fichier téléchargé. Pour y parvenir, nous avons initialisé une variable vide qui va contenir les éléments lus par le Reader. On vient ensuite vérifier si la ligne lue n'est pas vide. Si c'est le cas, on vient créer nos objets avec les données récupérées.

```

72     // Lecture du fichier OWD
73     String[] oneData = csvrOWD.readNext();
74     oneData = csvrOWD.readNext();
75     // Si la ligne lué n'est pas vide
76     while (oneData != null) {
77
78         // Lecture et ajout des données ligne par ligne
79         if (!((oneData[1] == null) || (oneData[1].length() == 0))) {
80             // Ajout du Continent
81             if (!(continentDAO.existsById(oneData[1]))) {
82                 Continent newContinent = saveContinent(oneData);
83                 continentDAO.save(newContinent);
84             }
85
86             // Ajout du Country
87             if (!(countryDAO.existsById(oneData[0]))) {
88                 Country newCountry = saveCountry(oneData, continentDAO);
89                 countryDAO.save(newCountry);
90             }
91
92             // Variables permettant de vérifier si l'InfoDailyCountry existe déjà
93             DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
94             LocalDate date = LocalDate.parse(oneData[3], formatter);
95
96             // Ajout de l'InfoDaily
97             if ((infoDailyCountryDAO.getIdInfoCountryByCountryInformedCodeCountryAndDate(oneData[0], date) == null)) {
98                 InfoDailyCountry newInfoDaily = saveInfoDailyCountry(oneData, countryDAO);
99                 infoDailyCountryDAO.save(newInfoDaily);
100            }
101        }
102    }
103    oneData = csvrOWD.readNext();
104}

```

verificateur()

Dans cette classe, nous avons également créé une méthode verificateur() qui prend en paramètre un attribut. Elle permet de convertir la chaîne de caractères récupérée en float, si celle-ci n'est pas null ou vide.

```

226     // Vérificateur donnée
227     public float verificateur(String attribute) {
228         float value = 0;
229         // On vérifie que la chaîne de caractère n'est pas nulle ou vide
230         if (!(attribute == null) || (attribute.length() == 0)) {
231             value = Float.valueOf(attribute);
232         }
233         return value;
234     }

```

Méthodes link

On vient ensuite lier un continent et un pays qui lui appartient ainsi qu'un pays et ses informations journalières.

```

160
161 // Liants
162 // Liant Continent - Country
163
164 public void linkContinentCountry(Continent continent, Country country) {
165     continent.getCountry().add(country);
166     country.setContinent(continent);
167 }
168
169 // Liant Country - InfoDailyCountry
170 public void linkCountryInfoDailyCountry(Country country, InfoDailyCountry infoDailyCountry) {
171     country.getInfo().add(infoDailyCountry);
172     infoDailyCountry.setCountryInformed(country);
173 }
...

```

Méthodes save

On passe ensuite à la sauvegarde de nos données dans les classes-entités. On indique tout d'abord pour chaque attribut de chaque classe-entité, quelle donnée du fichier lui correspond. On les ajoute ensuite dans un nouvel objet que l'on vient sauvegarder dans le DAO correspondant. Voici ce que cela a donné pour la classe-entité Country :

```

191
192 // Sauvegarde Country
193 public Country saveCountry(String[] oneData, ContinentRepository continentDAO) {
194     // Récupération des données
195     String codeCountry = oneData[0];
196     String nameCountry = oneData[2];
197
198     // Ajout des données
199     Country country = new Country();
200     country.setCodeCountry(codeCountry);
201     country.setNameCountry(nameCountry);
202
203     // Lien entre Continent et Country
204     Continent continent = continentDAO.findById(oneData[1]).get();
205     linkContinentCountry(continent, country);
206
207     // Actualisation de Continent
208     continentDAO.save(continent);
209
210     // Return
211     return country;
}

```

Pour la sauvegarde de la classe-entité InfoDailyCountry, c'est un peu différent. On vient vérifier les données qu'on récupère grâce à notre méthode verifyateur(). On les ajoute ensuite dans un nouvel objet. Puis, on vient regarder si la date du pays lié aux données journalières est égale à la date du jour moins 2 jours. Si c'est le cas, on vient vérifier tous les attributs de Country puis lui ajouter les nouvelles données journalières. On sauvegarde ensuite le pays dans le DAO. Voici le bout de code correspondant à la deuxième partie de la sauvegarde :

```

237     // Lien entre Country et InfoDailyCountry
238     Country country = countryDAO.findById(oneData[0]).get();
239     linkCountryInfoDailyCountry(country, infoDailyCountry);
240
241     // Réactualisation de Country
242     LocalDate today = LocalDate.now();
243     if (date.equals(today.minusDays(2))) {
244         // Récupération des données actuelles de Country
245         float icuPatients = verificateur(oneData[17]);
246         float hospPatients = verificateur(oneData[19]);
247         float totalTests = verificateur(oneData[25]);
248         float totalVaccinations = verificateur(oneData[34]);
249         float fullyVaccinated = verificateur(oneData[36]);
250         float stringencyIndex = verificateur(oneData[43]);
251         float population = verificateur(oneData[44]);
252         float gdp = verificateur(oneData[49]);
253
254         // Ajout des données
255         country.setTotalCases(totalCases);
256         country.setTotalCases(totalCases);
257         country.setTotalDeaths(totalDeaths);
258         country.setIcuPatients(icuPatients);
259         country.setHospPatients(hospPatients);
260         country.setTotalTests(totalTests);
261         country.setTotalVaccinations(totalVaccinations);
262         country.setFullyVaccinated(fullyVaccinated);
263         country.setStringencyIndex(stringencyIndex);
264         country.setPopulation(population);
265         country.setGdp(gdp);
266     }
267     // Actualisation de Country
268     countryDAO.save(country);
269
270     // Return
271     return infoDailyCountry;
272 }
```

Maintenant que nous avons abordé la partie “Model” ainsi que la classe permettant de récupérer les données et de les sauvegarder, nous allons présenter la partie “Controller”.

Controllers

Les “Controllers” ont plusieurs fonctions : associer des méthodes Java à des URL, contrôler les requêtes HTTP reçues, manipuler les objets du modèle en passant par les DAO et choisir la vue à afficher en fonction de la requête reçue, tout en leur transmettant les informations nécessaires.

Nous avons donc créé plusieurs Controllers. Le Controller ContinentController qui renvoie une page html et les autres contrôleurs qui renvoient des API contenant nos données au format json.

ContinentController

Le Controller ContinentController utilise le DAO ContinentRepository. Lorsque la requête HTTP “/continents/map” est faite, la méthode showContinentList() est exécutée et retourne la page html affichant nos cartes.

```

15  @Controller
16  @RequestMapping(path = "/continents")
17  public class ContinentController {
18
19      @Autowired
20      ContinentRepository continentDAO;
21
22      @GetMapping(path = "map")
23      public String showContinentList(Model model) {
24          model.addAttribute("continents", continentDAO.findAll());
25          return "maps";
26      }
27 }
```

ContinentApiController

Le Controller ContinentApiController utilise le DAO CountryRepository. Lorsque la requête HTTP “/api/continent/getContinent” est faite, la méthode getInfosContinent() qui prend en compte le nom du continent fait appel à la requête SQL de CountryRepository qui permet de créer l’API contenant toutes les informations pour chaque continent. On utilise l’interface InfoContinent pour que notre API affiche le nom des attributs et non leur emplacement.

```
15  @Service
16  @RequestMapping(path = "/api/continent")
17  public class ContinentApiController {
18
19      @Autowired
20      CountryRepository countryDAO;
21
22      // API renvoyant les infos actualisées d'un Continent
23      @GetMapping(path = "getContinent", produces = MediaType.APPLICATION_JSON_VALUE)
24      public @ResponseBody
25      InfoContinent getInfosContinent(@RequestParam(required = true) final String nameContinent) {
26          LocalDate today = LocalDate.now().minusDays(2);
27          return countryDAO.getInfosContinentByName(nameContinent, today);
28      }
}
```

CountryApiController

Le Controller CountryApiController utilise les DAO CountryRepository et InfoDailyCountryRepository. Lorsque la requête HTTP “/api/country/newCases” est faite, la méthode getNewCases() fait appel à la requête SQL de InfoDailyCountryRepository permettant de créer l’API contenant le nombre de nouveaux cas. Il s’agit de l’API utilisée par GeoChart Google pour afficher la carte.

La requête HTTP “/api/country/newDeaths” fait la même chose mais le nombre de nouvelles morts.

Les trois autres requêtes HTTP que nous avons font appel aux requêtes SQL de CountryRepository et permettent de créer les API permettant d’obtenir le nombre total de cas, le nombre total de morts et toutes les infos d’un pays. Voici les deux premières :

```
-->
27      // API utilisée par GeoChart Google
28      @GetMapping(path = "continent", produces = MediaType.APPLICATION_JSON_VALUE)
29      public @ResponseBody
30      List<Object> getNewCasesByContinent() {
31          LocalDate today = LocalDate.now().minusDays(2);
32          return infoDailyDAO.getNewCases(today);
33      }
34
35      // API utilisée pour afficher les infos d'un pays sélectionné
36      @GetMapping(path = "getCountry", produces = {MediaType.APPLICATION_JSON_VALUE})
37      public @ResponseBody
38      InfoCountry getCountryInfos(@RequestParam(required = true) final String nameCountry) {
39          LocalDate today = LocalDate.now().minusDays(2);
40          return countryDAO.getCountryByName(nameCountry, today);
41      }
42
```

InfoDailyCountryApiController

Le Controller InfoDailyCountryApiController utilise le DAO InfoDailyCountryRepository. Lorsque la requête HTTP “/api/infoDaily/country/totalstats” est faite, la méthode getAllDailyTotalStatsCountry() qui prend en compte le nom du pays fait appel à la requête SQL de InfoDailyCountryRepository qui permet de créer l’API contenant toutes les informations journalières d’un pays en fonction de la date.

La requête HTTP “/api/infoDaily/continent/totalstats” fait la même chose mais pour un continent. Et la requête HTTP “/api/infoDaily/world/totalstats” la même chose pour le monde entier.

Les trois dernières requêtes HTTP que nous avons faites permettent de créer les API contenant le nombre de nouveaux cas et de nouvelles morts par pays, continent ou pour le monde entier.

Voici nos trois premières requêtes :

```
--  
13  @Service  
14  @RequestMapping(path = "/api/infoDaily")  
15  public class InfoDailyCountryApiController {  
16  
17      @Autowired  
18      InfoDailyCountryRepository infoDailyDAO;  
19  
20      // API permettant d'afficher l'évolution des stats du Country en renvoyant une liste des infos journalières  
21      @GetMapping(path = "country/totalstats", produces = {MediaType.APPLICATION_JSON_VALUE})  
22      public @ResponseBody  
23      List<Object> getAllDailyTotalStatsCountry(@RequestParam(required = true) final String nameCountry) {  
24          return infoDailyDAO.getAllDailyTotalStatsByCountry(nameCountry);  
25      }  
26  
27      // API permettant d'afficher l'évolution des stats du Continent en renvoyant une liste des infos journalières  
28      @GetMapping(path = "continent/totalstats", produces = {MediaType.APPLICATION_JSON_VALUE})  
29      public @ResponseBody  
30      List<Object> getAllDailyTotalStatsContinent(@RequestParam(required = true) final String nameContinent) {  
31          return infoDailyDAO.getAllDailyTotalStatsByContinent(nameContinent);  
32      }  
33  
34      // API permettant d'afficher l'évolution des stats de World en renvoyant une liste des infos journalières  
35      @GetMapping(path = "world/totalstats", produces = MediaType.APPLICATION_JSON_VALUE)  
36      public @ResponseBody  
37      List<Object> getAllDailyTotalStatsWorld() {  
38          return infoDailyDAO.getAllDailyTotalStatsWorld();  
39      }  
..
```

WorldApiController

Le Controller WorldApiController utilise le DAO CountryRepository. Lorsque la requête HTTP “/api/world/getWorld” est faite, la méthode getInfosWorld() fait appel à la requête SQL de CountryRepository qui permet de créer l’API contenant toutes les informations du monde entier. On utilise l’interface InfoWorld pour que notre API affiche le nom des attributs et non leur emplacement.

```
23      @Autowired  
24      CountryRepository countryDAO;  
25  
26      // -- Nouvel ajout  
27      @GetMapping(path = "getWorld", produces = MediaType.APPLICATION_JSON_VALUE)  
28      public @ResponseBody  
29      InfoWorld getInfosWorld() {  
30          LocalDate today = LocalDate.now().minusDays(2);  
31          return countryDAO.getInfosWorld(today);  
32      }  
..
```

Nous allons maintenant vous présenter la partie “View” de notre application.

Views

Les “Views” permettent d'afficher les données du Model transmises par les Controllers. Elles sont gérées par un “moteur de vues” qui permet de définir la syntaxe à utiliser pour afficher les valeurs. Dans notre cas, nous avons utilisé Thymeleaf.

Nous avons deux pages HTML, index qui permet d'afficher notre page d'accueil et maps qui permet d'afficher les différentes visualisations graphiques. Ces deux pages ont été stylisées à l'aide des fichiers CSS style pour index et styleMaps pour maps. Nous avions d'ailleurs au départ fait une maquette de l'application, que vous pouvez trouver en annexe (page 22).

maps.js

Pour les différentes fonctionnalités liées à la carte, nous avons créé un fichier Javascript maps. Tout d'abord nous avons créé une table de correspondance pour associer la bonne carte au nom de la zone concernée. Ensuite nous lançons l'API Google permettant de charger les maps. Et on vient faire l'appel AJAX dès le chargement de la page.

```
1 // Table de correspondance qui associe la bonne carte au nom de la zone
2 const mapIdContinents = [['World', 'world'], ['Africa', '002'], ['Europe', '150'], ['America', '019'], ['Asia', '142'], ['Oceania', '009'], ['North America', '021'], ['South America', '005']];
3
4 // Lancement de l'API Google permettant de charger les maps
5 google.charts.load('current', {'packages': ['geochart'], 'mapsApiKey': 'AIzaSyD-9tSrke72PouQmMX-a7eZSW0jkFNBWY'});
6
7 // On fait l'appel AJAX dès le chargement de la page
8 google.charts.setOnLoadCallback(getRegionsInfo);
```

Dans ce fichier, nous avons tout d'abord une fonction permettant de retourner la carte associée au nom du continent.

```
10 // Fonction qui retourne la carte associée au nom du continent
11 function getMap(nameContinent) {
12     for (let mapContinent of mapIdContinents) {
13         if (nameContinent == mapContinent[0]) {
14             return mapContinent[1];
15         }
16     }
17     return ['World', 'world'];
18 }
```

Ensuite nous faisons une requête AJAX getRegionsInfo() qui permet de récupérer les données à afficher en faisant la requête HTTP associée à CountryApiController. On indique que si la requête excelle alors on exécute la fonction drawRegionsMap(), sinon on exécute la fonction showError() qui va afficher un message d'erreur.

```

21 // Requête AJAX qui récupère les données à afficher
22 function getRegionsInfo() {
23     var nameAttribute = document.getElementById("nameAttribute").value;
24     $.ajax({
25         type: "GET",
26         url: "/api/country/" + nameAttribute,
27         dataType: "json",
28         contentType: "application/json",
29         success: drawRegionsMap,
30         error: showError
31     });
32 }

```

Notre fonction `drawRegionsMap()` permet d'afficher les cartes. Tout d'abord, on initialise les colonnes puis on ajoute chaque ligne de l'API dans une table de données. Ensuite nous faisons appel à la fonction permettant de récupérer l'ID des maps. Enfin, pour chaque pays affiché, on ajoute un événement permettant de lire ses infos en faisant la requête HTTP associée à `CountryApiController`. Si la requête excelle, on exécute la fonction `showInfoCountry()`, sinon on affiche un message d'erreur. On finit par afficher la carte.

```

33 // Function permettant d'afficher les cartes
34 function drawRegionsMap(result) {
35     // Initialisation des colonnes
36     var headers = [["Country", "New Cases"]];
37
38     // On ajoute chaque ligne de l'API dans une table de données
39     for (let i = 0; i < result.length; i++) {
40         headers.push(result[i]);
41     }
42
43     var dataTable = google.visualization.arrayToDataTable(headers);
44     var chart = new google.visualization.GeoChart(document.getElementById('region_div'));
45     var options = {
46         // On fait appel à la fonction permettant de récupérer l'id des maps
47         region: getMap(document.getElementById("nameContinent").value),
48         enableRegionInteractivity: true,
49         colorAxis: {colors: ['rgb(29, 66, 115)']},
50         backgroundColor: {colors: ['transparent']}};
51
52     // Pour chaque pays affiché, on ajoute un événement afin de lire ses infos
53     google.visualization.events.addListener(chart, 'select', function () {
54         var selectedItem = chart.getSelection()[0];
55         if (selectedItem) {
56             var country = dataTable.getValue(selectedItem.row, 0);
57             getCountryStats(country);
58             $.ajax({
59                 type: "GET",
60                 url: "/api/country/getCountry?nameCountry=" + country,
61                 dataType: "json",
62                 contentType: "application/json",
63                 success: showInfoCountry,
64                 error: showError
65             });
66         }
67     });
68 }

```

La fonction `showInfoCountry()` permet d'afficher les informations d'un pays. Pour cela, on rend le tableau visible puis on met en titre du tableau le nom du pays. On finit par afficher les infos avec la fonction `showInfo()`.

```

104 // Fonction permettant d'afficher les infos d'un Country
105 function showInfoCountry(result) {
106     // On rend le tableau visible
107     var table = document.getElementById("tableInfo");
108     table.style.display = "initial";
109
110    // On met en titre du tableau le nom du Country
111    var country = document.getElementById("title");
112    country.innerHTML = result.name;
113
114    // On affiche les infos
115    showInfo(result);
116}

```

Cette fonction `showInfo()` permet d'afficher les infos. Pour cela on rend le tableau visible et on récupère et affiche nos informations.

```

131 // Fonction permettant d'afficher les infos des continents
132 function showInfo(result) {
133     // On rend le tableau visible
134     var table = document.getElementById("tableInfo");
135     table.style.display = "initial";
136
137    // On récupère et affiche les infos
138    var population = document.getElementById("population");
139    population.innerHTML = result.pop.toLocaleString();
140    var cases = document.getElementById("cases");
141    cases.innerHTML = result.tcases.toLocaleString();
142    var newCases = document.getElementById("newCases");
143    newCases.innerHTML = result.ncases.toLocaleString();
144    var deaths = document.getElementById("deaths");
145    deaths.innerHTML = result.tdeaths.toLocaleString();
146    var newDeaths = document.getElementById("newDeaths");
147    newDeaths.innerHTML = result.ndeaths.toLocaleString();
148    var vaccinations = document.getElementById("vaccinations");
149    vaccinations.innerHTML = result.tvaccinations.toLocaleString();
150    var vaccinations2nd = document.getElementById("vaccinations2nd");
151    vaccinations2nd.innerHTML = result.fvaccinated.toLocaleString();
152    var newVaccinations = document.getElementById("newVaccinations");
153    newVaccinations.innerHTML = result.nvaccinations.toLocaleString();
154
155}

```

Nous avons également une fonction `getMapsInfo()` qui permet d'afficher les infos des continents lorsque l'on sélectionne leur carte. Pour cela on vérifie que la carte sélectionnée n'est pas celle du monde en faisant la requête HTTP associée à `ContinentApiController`. Si la requête excelle, on exécute la fonction `showInfoMap()`.

```

77 // Fonction permettant d'afficher les infos des continents quand on sélectionne leur map
78 document.getElementById("nameContinent").addEventListener("change", getMapsInfo());
79 function getMapsInfo() {
80     var map = document.getElementById("nameContinent").value;
81
82    // On vérifie que la map sélectionnée n'est pas celle de World
83    if (map != 'World') {
84        $.ajax({
85            type: "GET",
86            url: "/api/continent/getContinent?nameContinent=" + map,
87            dataType: "json",
88            contentType: "application/json",
89            success: showInfoMap,
90            error: showError
91        });
92    } else {
93        $.ajax({
94            type: "GET",
95            url: "/api/world/getWorld",
96            dataType: "json",
97            contentType: "application/json",
98            success: showInfoMap,
99            error: showError
100       });
101   }
102}

```

Cette fonction `showInfoMap()` permet d'afficher les infos d'un continent. Pour cela on rend le tableau visible, on met en titre du tableau le nom du continent et on affiche les infos avec la méthode `showInfo()`.

```

117
118  function showInfoMap(result) {
119      // On rend le tableau visible
120      var table = document.getElementById("tableInfo");
121      table.style.display = "initial";
122
123      // On met en titre du tableau le nom de la Map
124      var country = document.getElementById("title");
125      country.innerHTML = document.getElementById("nameContinent").value;
126
127      // On affiche les infos
128      showInfo(result);
129  }

```

graphs.js

Pour les différentes fonctionnalités liées au graphique, nous avons créé un fichier `graphs`. Dans celui-ci, nous commençons par lancer l'API de Google permettant de charger les graphes.

Ensuite, nous avons une fonction `getCountryStats()` qui permet de récupérer la liste des infos journalières d'un pays. Pour cela, on fait la requête HTTP associée à `InfoDailyCountryApiController`. Si celle-ci excelle, on initialise le tableau de données et les colonnes et pour chaque ligne de l'API, on transforme les objets de la première ligne en date. On trace ensuite le graphique correspondant.

```

4     // Fonction permettant de récupérer la liste des infos journalières d'un Country
5  function getCountryStats(country) {
6      $.ajax({
7          type: "GET",
8          url: "/api/infoDaily/country/stats?nameCountry=" + country,
9          dataType: "json",
10         contentType: "application/json",
11         // Fonction permettant d'afficher les infos dans le graphe
12         success: function (result) {
13             // On initialise le tableau de données ainsi que les colonnes
14             var dataTable = new google.visualization.DataTable();
15             dataTable.addColumn('date', 'Date');
16             dataTable.addColumn('number', 'Total Cases');
17             dataTable.addColumn('number', 'Total Deaths');
18
19             // Pour chaque ligne de l'API, on transforme les objets de la 1ère ligne en date
20             for (let line of result) {
21                 line[0] = new Date(line[0]);
22             }
23             dataTable.addRows(result);
24
25             var options = {
26                 title: country,
27                 curveType: 'function',
28                 legend: {position: 'bottom'}
29             };
30             var chart = new google.visualization.LineChart(document.getElementById('graphe'));
31
32             // On trace le graphe
33             chart.draw(dataTable, options);
34         },
35         error: showError
36     });

```

Notre fonction `getAreasStats()` permet de récupérer les infos journalières d'un continent. On vérifie alors si la map sélectionnée n'est pas celle du monde et on fait la requête HTTP associée à `InfoDailyCountryApiController`. Si celle-ci excelle, on exécute la fonction `showStatsMap()`.

```

41 // Fonction permettant de récupérer les infos journalières d'une map
42 function getAreaStats() {
43     var map = document.getElementById("nameContinent").value;
44
45     // On détecte si la map sélectionnée est World
46     if (map != 'World') {
47         $.ajax({
48             type: "GET",
49             url: "/api/infoDaily/continent/stats?nameContinent=" + map,
50             dataType: "json",
51             contentType: "application/json",
52             success: showStatsMap,
53             error: showError
54         });
55     } else {
56         $.ajax({
57             type: "GET",
58             url: "/api/infoDaily/world/stats",
59             dataType: "json",
60             contentType: "application/json",
61             success: showStatsMap,
62             error: showError
63         });
64     }

```

Cette fonction `showStatsMap()` permet de tracer les graphes d'une map. Pour cela, on initialise le tableau de données et les colonnes. Pour chaque ligne de l'API, on transforme les objets de la première ligne en date et enfin, on trace le graphique correspondant.

```

67 // Fonction permettant de tracer les graphes d'une map
68 function showStatsMap(result) {
69     // On initialise le tableau de données ainsi que les colonnes
70     var dataTable = new google.visualization.DataTable();
71     dataTable.addColumn('date', 'Date');
72     dataTable.addColumn('number', 'Total Cases');
73     dataTable.addColumn('number', 'Daily Cases');
74
75     // Pour chaque ligne de l'API, on transforme les objets de la 1ère ligne en date
76     for (let line of result) {
77         line[0] = new Date(line[0]);
78     }
79     dataTable.addRows(result);
80
81     var options = {
82         title: document.getElementById("nameContinent").value,
83         curveType: 'function',
84         legend: {position: 'bottom'}
85     };
86     var chart = new google.visualization.LineChart(document.getElementById('graphe'));
87
88     // On trace le graphe
89     chart.draw(dataTable, options);
90 }

```

IV) Pour aller plus loin

Bien évidemment, notre application peut être améliorée.

Tout d'abord, il serait préférable qu'elle soit responsive, c'est-à-dire qu'elle puisse s'adapter correctement en fonction de la taille de l'écran.

On pourrait également y ajouter un calendrier pour que les utilisateurs puissent choisir le jour qui les intéresse et ainsi afficher les données correspondantes.

De plus, il serait intéressant que l'on puisse zoomer sur les zones qui nous intéressent. Comme par exemple, faire un zoom sur la France et pouvoir afficher des données pour chaque département.

V) Gestion de projet

Du point de vue de la gestion de projet, nous avons utilisé plusieurs outils pour collaborer au mieux ensemble sur ce projet.

Pour réaliser notre diagramme UML nous avons utilisé la plateforme Lucidchart qui permet de créer des diagrammes en ligne. Ceci était donc beaucoup plus facile pour nous afin que chacun puisse modifier le diagramme et visualiser ce que les autres faisaient en temps réel.

Pour avoir une visualisation globale des tâches à réaliser tout au long du projet, nous avons utilisé Trello qui est un outil collaboratif basé sur la méthode Agile. Ainsi, les tâches étaient classées en plusieurs catégories en fonction de leur état : à faire, en cours, fait, vérifié.

Avec les restrictions actuelles, nous avons énormément dû travailler à distance. Pour faciliter nos échanges, nous avons donc utilisé l'application de messagerie instantanée Messenger ainsi que l'application de vidéoconférence Zoom qui nous permettait facilement de partager notre écran.

Annexes

Diagramme UML initial

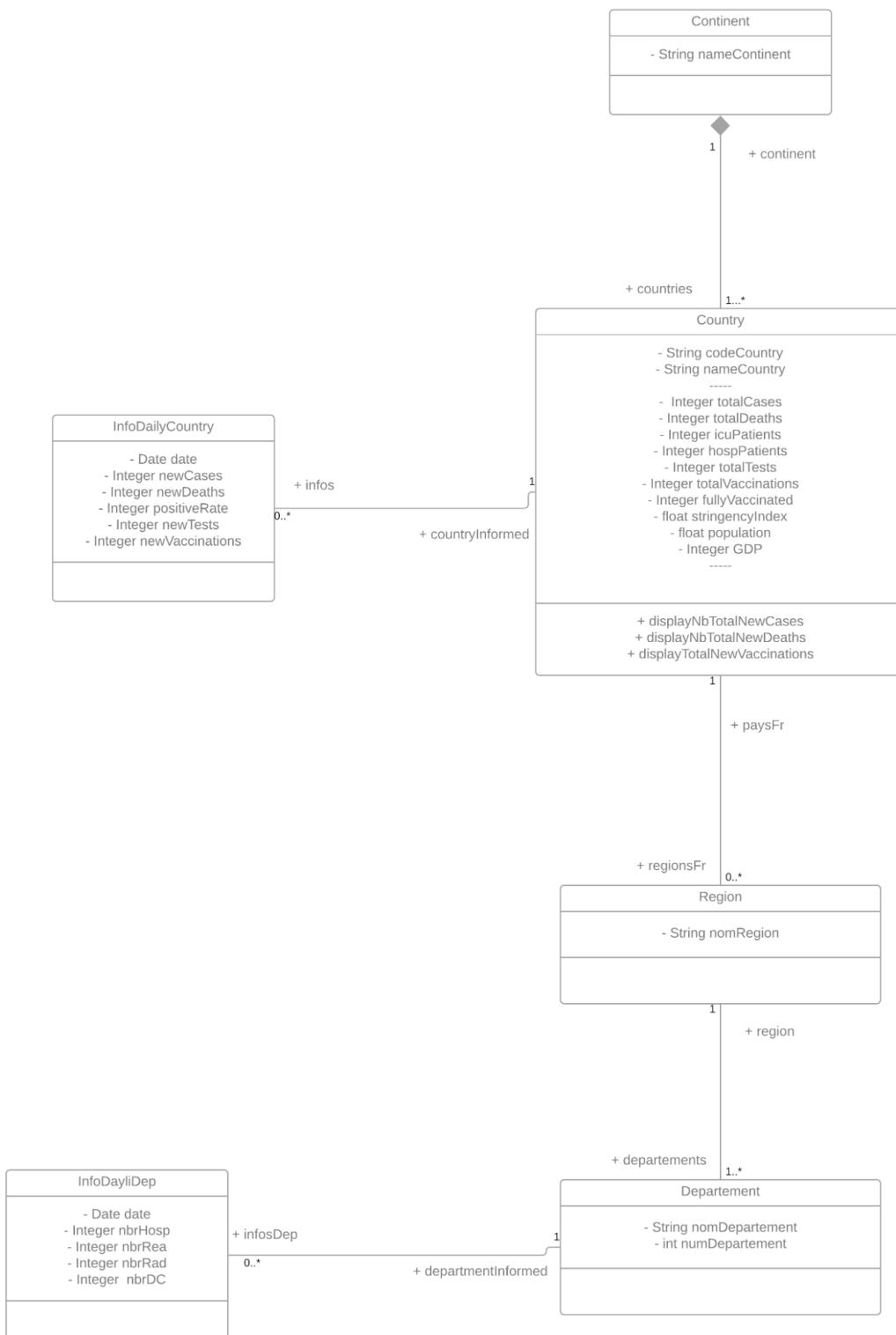
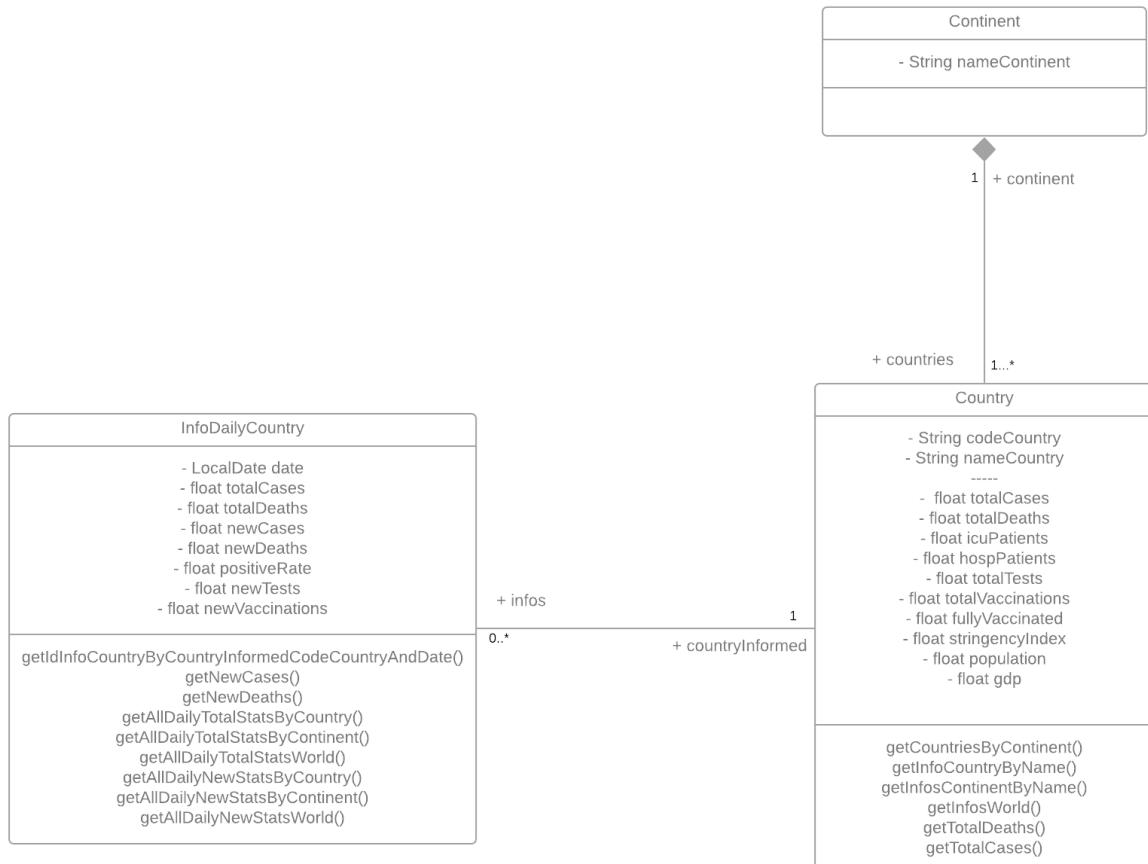


Diagramme UML final



Maquette

