METEORITES REST API

This project has been developed for the Data Access subject of the second course of Development of Multiplatform Applications in Uni Eibar. The main task is to develop a Rest Api that accesses a MongoDB database and to learn how a Rest Api can fit in a web application. In this case a JSON dataset was selected and loaded in a MongoDB database. That dataset collects all the meteorites that have fallen to earth in the last 200 years. These are the general use steps to take into account:

- Swagger access: http://localhost:8081/swagger-ui/index.html
- MongoDB connection String: mongodb://localhost
- Database name: meteorites
- Collection name: fallen
- Maven execution command (same directory as pom.xml): mvn spring-boot:run

DATASET

The dataset has a particular structure, each meteorite is a JSONObject but at the same time these meteorites have their own attributes; nine Strings and an JSONObject that inside has an array. The main idea was to deal with a certain level of depth in our datasets avoiding picking plain datasets. The dataset selected for the project was picked from a collection of datasets located in GitHub.

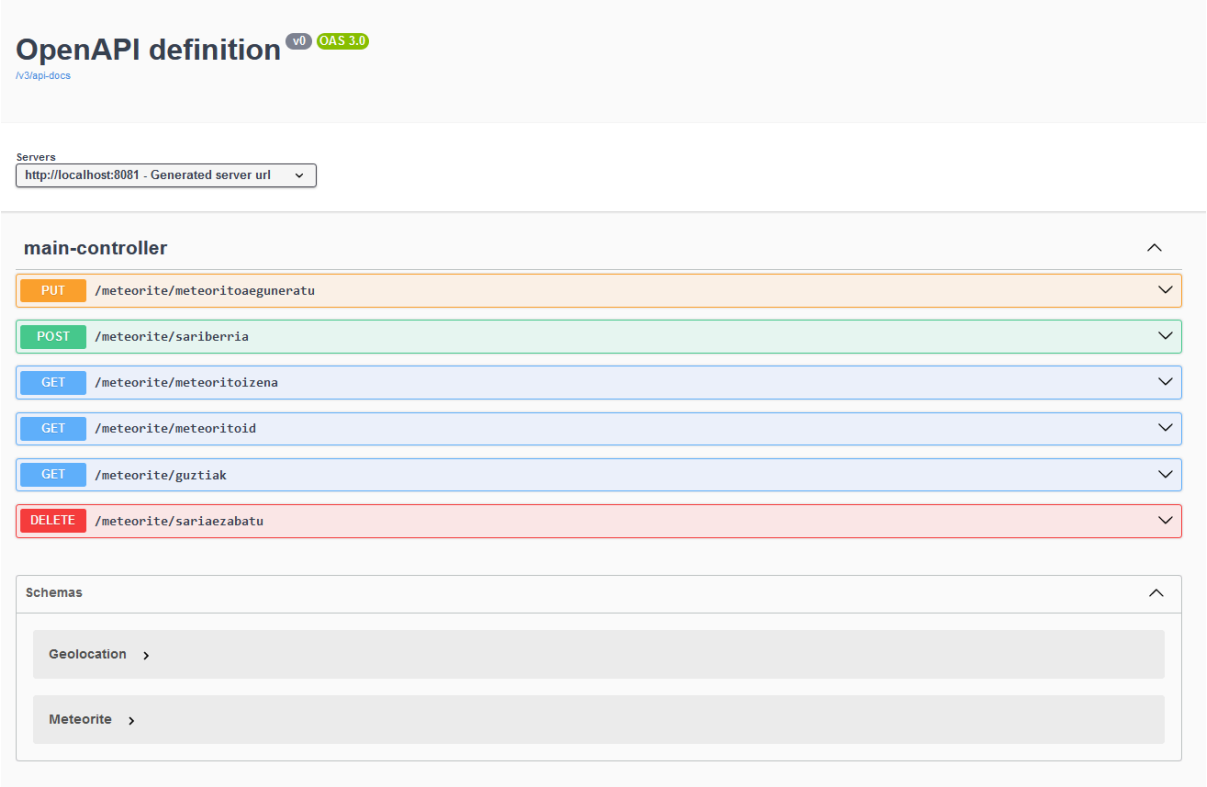
This is the link: https://github.com/jdorfman/awesome-json-datasets?tab=readme-ov-file

As mentioned before the dataset was originally taken as a JSON document. Before the dataset was imported to a MongoDB database it suffered a small change. The attribute called id was changed to idn because it can be confused by mongo as the object id. To avoid problems we changed that.

Before starting making the import is compulsory to create a new connection to our MongoDB server. In this case we configured a connection to the localhost using this connection string: mongodb://localhost.

REST SERVICE

The Rest service has a particular structure. Using this service the user can make different types of requests, those types are the same that a CRUD app makes; create(POST), read(GET), update(PUT) and delete(DELETE). Each endpoint manages a different type of request, that's why depending which one is going to be executed the service will do one type of operation or another. The request can be made using the Swagger user interface or another development environment as Insomnia, Postman etc. From Swagger you can access the different endpoints, see the model schemas and get the api doc in yaml format. This is an overview:



GET -> /meteorite/guztiak -> This returns a JSON with all the meteorites

GET -> /meteorite/meteoritoid -> This return a JSON of the exact meteorite

GET -> /meteorite/meteoritoizena -> This return a JSON of the exact meteorite

POST -> /meteorite/metberria -> Meteoritoa era egokian gehitu da

PUT -> /meteorite/meteoritoaeguneratu -> Response headers and response (ResponseEntity HttpRequestCode)

DELETE -> /meteorite/metezabatu -> Meteoritoa era egokian ezabatu da

IMPORTANTE: ESTOS ENDPOINT METELOS DENTRO DE UNA TABLA DE MARKDOWN, ES FACIL HACER

MONGODB

This SpringBoot application is a service that creates requests for a MongoDB database. For creating the database it is neccesary to install the Community edition of MongoDB and MongoDB Compass, which is a MongoDB client. Once installed we open the client and we create a connection to our local machine. Here you can create remote connections to servers, to the local machine and even to the MongoDB cluster or the Cloud.

Editing and adjusting that connection string will give you the possibility to be able to connect where you need. In this case the application is configured to connect to the local machine via Localhost. Entering mongodb://localhost you can connect and proceed to create the database.

In the left side you can see the databases, there you create the database and the collection where the data is going to be kept. The name of the database is called meteorites and the collection is called fallen. Once the database and the collection are created we import our dataset. The dataset is a JSON document, just importing the document mongo will create a large amount of documents. Each document is a meteorite and each meteorite has its own attributes:

{

"idn": "446",

"name": "Al Rais",

"nametype": "Valid",

"recclass": "CR2-an",

"mass": "160",

"fall": "Fell",

```
 "year": "1957-01-01T00:00:00.000",
 "reclat": "24.416670",
 "reclong": "39.516670",
 "geolocation": {
  "type": "Point",
  "coordinates": [
   39.51667,
   24.41667
  ]
 }
```


JAVA PROJECT

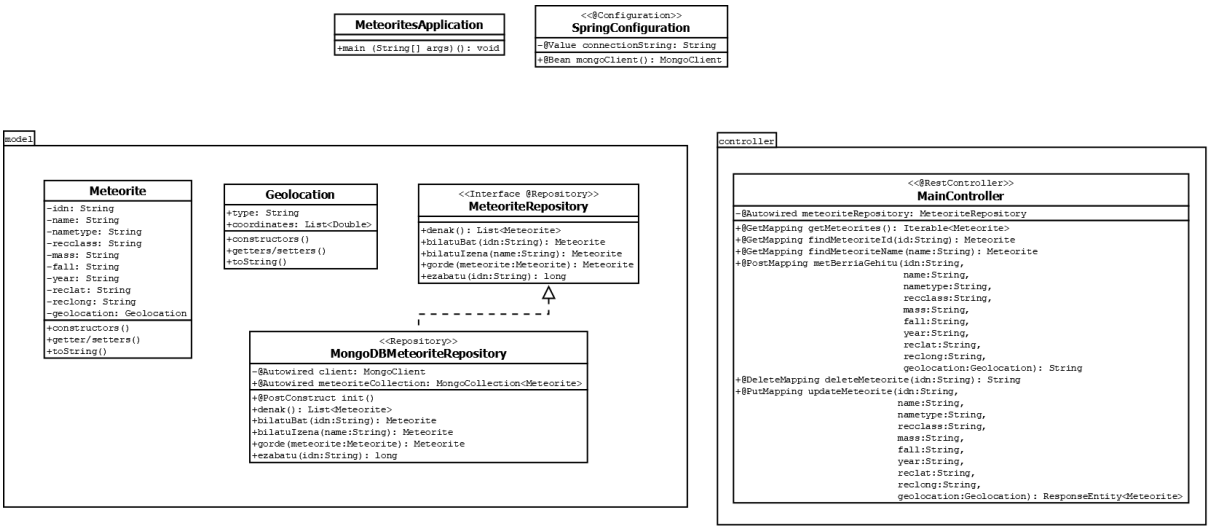
The Java Project has been developed creating a SpringBoot project. We are also using a Maven type of project because we can comfortably add dependencies and include them in our project. This is made using the pom.xml document inside the project. Inside the pom we can find the build in, plugins and dependencies of our project. Is a effective and comfortable way to add different types of resources to our project.

The SpringBoot type of projects have a particular structure of classes and models. The project is divided in two main packages; model and controller. Outside those packages we have two classes more. The first class is the SpringConfiguration. With this class the application can get the connection string from the application properties and get the MongoDB client creating at the same time the conversion from BSONDocument to JavaObject. And the second class is the Application main class. With this class the application gets started and runs the whole service.

Inside the model package we can find different types of classes. The object models are one of those, with these models the application creates a mapping to structure the data and be able to process them in MongoDB. As we are working with a type of object where the object itself has an object inside we have to map also the subobject. In this case we have a Meteorite class and inside that Meteorite class we have as an attribute the geolocation that is an Object itself and inside contains a list. For that purpose the Geolocation class has been developed also having its own attributes. Once these models are built the conversion from the JavaObjects and the Mongo documents can be made. Executing Swagger and scrolling down to the bottom we can see the schemas.

Lefting the objects classes apart, we have two classes more that are the repositories. In those repositories we can collect the data and then manage them to execute operations. We have an interface where we declare the basic functions and then we have a MongoDB repository. This repository implements the functions declared in the repository interface. Those functions are the basic operation that later the controller will use to create different types of requests.

Finally, inside the controller package we have a single class. This class is the main controller where we create the different endpoints of the service and the main request mapping. This is the class diagram of the application:



CONVERSOR

This program is a conversor and importation and exportation program. Is able to make different types of operations based on documents that we are going to add in the data directory. The program has been developing using Java building a maven type project. Inside the model directory we have three object models, one for the Geolocation that is an attribute of the main Meteorite class, another for the Meteorites class that is a list of the main Meteorite class and finally the Meteorite class keeping the attributes of the main class. All the models have been annotated with JAXB annotation to be able to convert to XML.

This program has as a start point a JSON document. We have a functionality to export a JSON document from the data we have hosted in our MongoDB database. The document will be created in the data directory and it will be the main document, we can also add our own JSON documents but the data that is stored in the documents has to be mapped to use the structure of our main Meteorite class.

This program has been designed to optimize the task of conversion of our own data. Is important to mark that the base is a JSON file that is going to be exported from our database directly. The main task is based on working with that data instead of creating a simple format conversor. Based on the data we can also convert to CSV and XML but we have to take into account that all data is going to be limited to our main class structure. We can also add new data to a JSON document and then import to mongo, but as said before that data has to be added using a concrete structure and written on a JSON document that is one of the main formats that Mongo can read and work with.

These are the main transactions that the program can make:

EXPORT FROM MONGODB -> This functionality is going to be able to export all data to a JSON file. First the program makes a connection to the database specifying the connection string, the database name and the collection name. Once we make the connection the program will ask the user to enter a name for the JSON file. After that the functionality will extract all the data to the file and will be stored in the data directory.

IMPORT TO MONGODB -> MongoDB is able to import data in two main formats using the import data function, those formats are CSV and JSON. As we have an initial JSON document to build our database we have specified JSON as our default format. That's why this functionality will take as base a JSON document and will be able to import the data to mongo. The functionality will ask the user to specify the name of the JSON document and will create a connection. After that will call a

function and this function will return a list of Meteorite objects. Then it will transform from Meteorite list to a Document list and will be inserted on the collection of MongoDb.

CONVERT TO CSV -> This functionality is going to be able to write a CSV file, to do that first we extract a Meteorite list that is going to be sent calling to a function. Then we will take that list and we are going to be able to write the CSV file. First we ask the user to write the name of the JSON document where we are going to extract the list and then we will ask the user to enter a name for the new CSV file. Once this is specified the program will write the names of the fields and will start to extract the data from each Meteorite object. The geolocation attribute is a bit special because in some cases it can be null, so first we make the comparison of whether it is null or not. Once the conprobation is done the program will write all the data and the file will be created in the data directory.

CONVERT TO XML -> This functionality is going to be able to write an XML file. As the program does with the CSV converter it can also do the same but writing an XML file. The operation flows in the same way, first we call a function and we will get a Meteorite list that is going to be the data that we want to convert, and once with that list we create a Marshaller and we create a XML document passing the list and an OutputStream. When the functionality is going to be executed the program ask the user to enter a name for the XML file, then will do the operation and the file will be created in the data directory.