Cerebro Tutorial

This tutorial will give you an idea on how to integrate Cerebro features into your existing applications. We prepare a simple web application, namely "**blankapp**", which allows users to search for movie titles and rate a movie. At the end of the tutorial, you should able to understand the necessary steps to integrate Cerebro into the "**blankapp**" to create new recommendation features such as movies recommendation.

This tutorial includes 3 parts:

1. Part 1: Setting up the **blankapp**
2. Part 2: Explaining Cerebro's database
3. Part 3: Integrating Cerebro's item recommendation feature.

Dependencies. The "**blankapp**" application is consisted of a frontend (implemented by ReactJS), a backend (Java – Spring boot framework) and a database (mongoDB).

Don't worry, you don't need to know about ReactJS as it is solely for demonstration purposes. You will, however, be expected to know the basic of relational/non-relational database (more about this in part 2) and Java as it is the core language in which Cerebro is implemented.

First of all, clone the tutorial package:

```
git clone https://github.com/PreferredAI/cerebro-tutorial.git
```

Part 1: Blankapp – A web app for movie rating recording.

In order to demonstrate Cerebro's recommendation features we will need a place to show the result of its inner working. Towards this end, we will set up a web page for movie rating with the following functionalities:

- We allow users to log in based on their ids
- Once logged in, user can search for existing movies in the database.
- User can click on a movie to rate that movie and/or see the rating score he/she gave that movie the last time.

- **Setting up database for "blankapp"**

We'll start with setting up the database for our application. We will use mongoDB, so if you haven't installed mongoDB on your machine, click on the link below to install and get an instance of mongoDB running on your machine:
Installation: https://docs.mongodb.com/manual/installation/

Note: you are also recommended to install also MongoDB Compass for ease of control your database.

Next we will use the dataset **movieLens 1M**[1] to fill in our database.

Please refer to the README file in the **ml-1m** folder to better understand the format of the files as well as type of information this dataset provides.

[1] Source: http://files.grouplens.org/datasets/movielens/ml-1m.zip

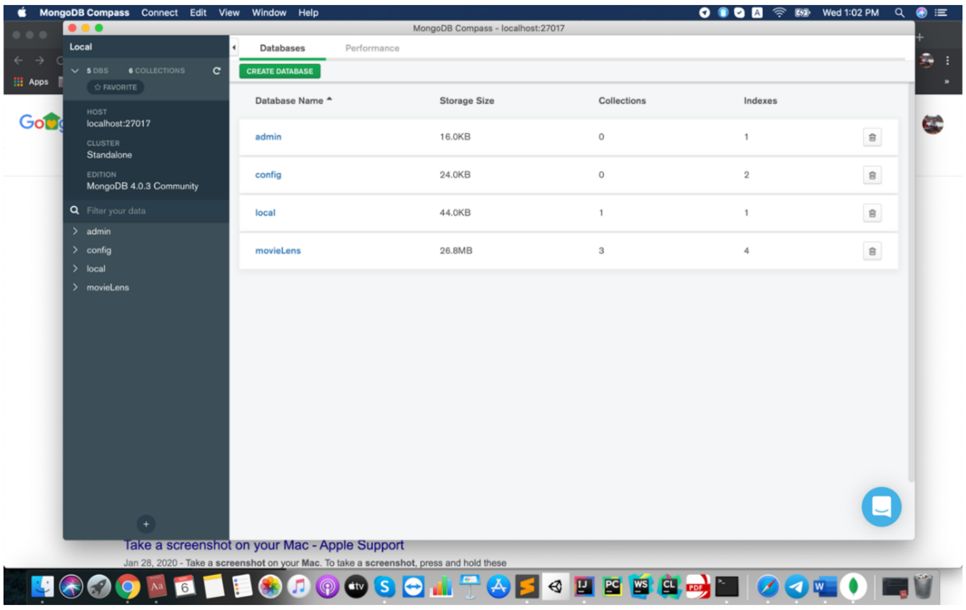
Inside the folder **extractMovieLens** of the tutorial package is a jar file that will help you to extract **movieLens-1M** into your mongoDB (make sure you are running an instance of mongoDB in the background):

Then navigate your terminal to the **extractMovieLens** folder, then type:

```
java -jar extract-standalone.jar -filePath $path_to_movielens_folder$
```

By default, the extract.jar file will assume that you are running your mongoDB on localhost:27017 (which is also the default of mongodb). If you are running mongoDB differently, add the connection information with flags -host and -port into the above command line.

With that, now open MongoDB Compass. Click on "CONNECT" if you are doing everything by the default. It should look like this:



HOST
localhost:27017

CLUSTER
Standalone

EDITION
MongoDB 4.0.3 Community

Database Name	Storage Size	Collections	Indexes
admin	16.0KB	0	1
config	24.0KB	0	2
local	44.0KB	1	1
movieLens	26.8MB	3	4

Now click on the movieLens database. You can see that all of the information from the movieLens files has been imported by the jar file into 3 collections:

- items
- ratings
- users

With that you have finished setting up the underlying database for the **blankapp** web application.

- **Frontend for "blankapp"**

Next let's open the **blankapp-frontend** project in the tutorial package.

Go to the link below to install Node if you haven't:

https://nodejs.org/en/download/

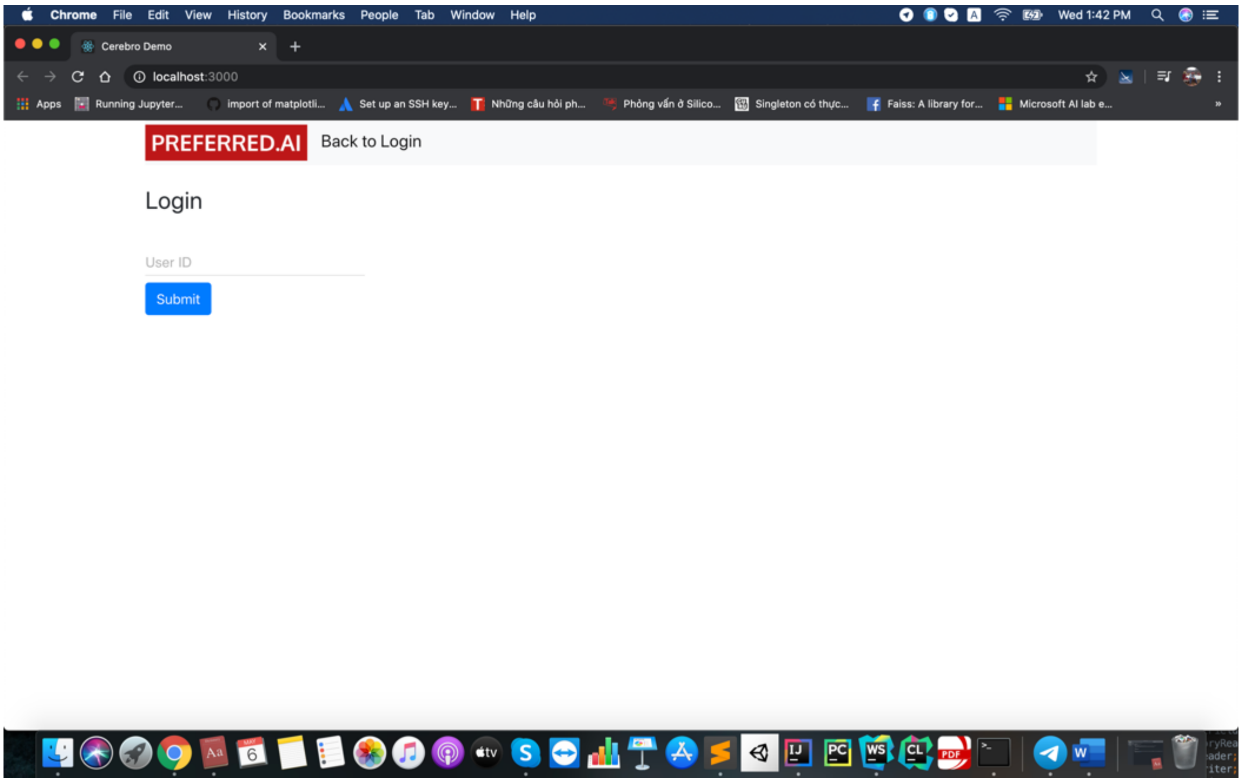
Navigate to the project folder you just cloned in your terminal and type:

```
npm install
```

Wait till everything is finished and type:

```
npm start
```

It should look like this your browser:



- **Backend Server for “blankapp”**

However, our backend isn’t there yet so you shouldn’t click on anything right now.

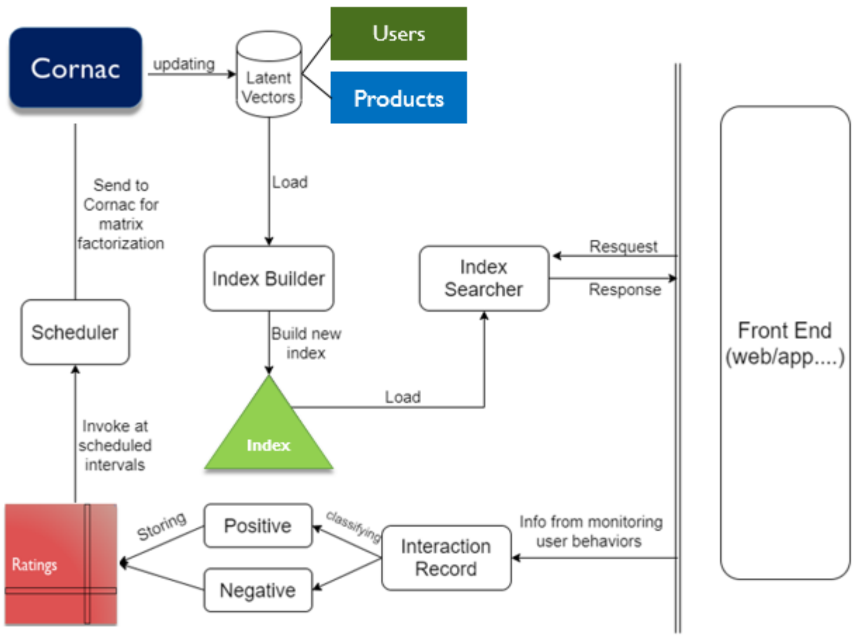
Navigate to the **blankserver** folder in the tutorial package:

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

Note: if you are running mongoDB differently than the default setting, please pass these setting into the file **application.properties** in the project before running the above command line.

With that you have successfully set up the web application, feel free to interact with it. Once you are ready to move on to the next part hit ctrl + C in the **blankserver** terminal.

Part 2: Explaining Cerebro ’s database requirements:



Cerebro is a rating-based recommendation library, it generates recommendations based on score calculated from some customizable scoring functions (cosine similarity, Euclidean distance,...etc) on two vectors. However, as you might be wondering what these two vectors represent, one of them represents the user preferences and the other the properties of the item; they are generated by our in-house library - Cornac. Thus we have to make sure Cornac is working properly in order for Cerebro to function.

The simplest way to summarize Cornac inner working is that it takes in a list whose rows in the format {userID, itemID, rating_score}. And then for each unique userID and itemID in that list it associates with a feature vector such that the higher similarity score between them the more likely the corresponding user and item are to match.

Therefore, Cerebro has to collect data in a format that can easily fuels its internal Cornac feature vector generator. In short, any applications that use Cerebro will be expected to provide followings upon the first time use:

+ A list where each element is the unique id (String typed) of a user in your database.

+ A list where each element is the unique id (String typed) of an item in your database.
+ A list where each element is the tuple {userID, itemID, rating_score, {time stamp}}.

Every elements in these three lists should be unique. You should have no problem with this requirement on the userID list and the itemID list, but you may run into the problem that one user may rate an item differently at different point of time. After all, human feeling about a certain thing rarely remains the same through time, and taking a look at the ratings collection of the database you extracted in part one you will also notice there is time field when the rating happened.

In this tutorial we will always picks the latest one if there is more than one tuple with the same {userID, itemID} signature.

Part 3: Integrating Cerebro's item recommendation feature:

First let's get Cerebro, clone Cerebro if you haven't already:

```
git clone https://github.com/PreferredAI/cerebro.git
```

Navigate into the folder **cornac_webservice** in the tutorial package, and type the following command:

```
python main.py
```

That should get an instance of flask webserver up and running to take care of the vectors generation.

Next, let's take a look at our backend – blankserver. Navigate you way to the file AppController.java, you will be working mostly with this file to bring out the recommendation feature and show it to your frontend.

Pay close attention to the last 3 functions in this file:

```
@RequestMapping(value = "/searchTitle", method = RequestMethod.POST)
public ItemListResponse searchKeyword(@Valid @RequestBody TextQuery qObject) throws
IOException
```

```
@RequestMapping(value="/getRating", method = RequestMethod.POST)
public RatingResponse getRating(@Valid @RequestBody PairIds pairIds) throws
IOException
```

```
@RequestMapping(value="/setRating", method = RequestMethod.POST)
public void setRating(@Valid @RequestBody Rating rating) throws IOException
```

So functions **searchKeyword()**, **getRating()**, **setRating()** respectively. Recall that on the frontend, through our GUI, we allow user 3 kinds of action:

- Search for a movie by its name which is handled by **searchKeyword()**,
- After searching you can click on an item from the list retrieved, to see your past rating on the movie or whether you have rated that movie or not (**getRating()**).
- You can rate on a movie which is handled by **setRating()**. In the case you have already rated the movie last time, the next time you click on this movie it will show your latest rating. All of your past ratings are still stored in the database.

Also this line at the top of the file:

```
@RequestMapping("/blankapp")
```

This mean that you are mapping, for example, a call to function **searchKeyword()** to a HTTP request to the url:
http://$your_host_name$:$your_port_number$//blankapp/searchTitle

In this case your server is running on your machine – thus localhost, and the default port is 8080. So to call the function **searchKeyword()** from a different application (our front end) we just need to make a http request to the url :

http://localhost:8080//blankapp/searchTitle

Of course you still have to fill the body of that request with a query string. But for now you have understood how to map a function on the server side to a url link.

Next we will first focus on exporting our current database to Cerebro's database. Inside the **blankserver** there is two files Exporter.java and MongoExporter.java that has already been implemented for you. These class will be in charge of exporting your current database to cerebro's database. Now let's expose this service through an url call.

We will implement a function that will be mapped to a URL link just like what you just learn above, this function will be called from the cerebro server so that it could import your database into its database.

Copy this code into your AppController class in blankserver project:

```
@CrossOrigin
@RequestMapping(value="/export", method = RequestMethod.POST)
public void export(@Valid @RequestBody ConnectInfo info){
    Exporter exporter = new MongoExporter(usersRepository,
            itemsRepository,
            ratingCollection, info.dbhost, info.dbport);
    exporter.exportUserIds();
    exporter.exportItemIds();
    exporter.exportRatings();
}
```

Once cerebro started up it will check if its dedicated database is filled or not then. If not, it will make a POST request to this function we just implement here.

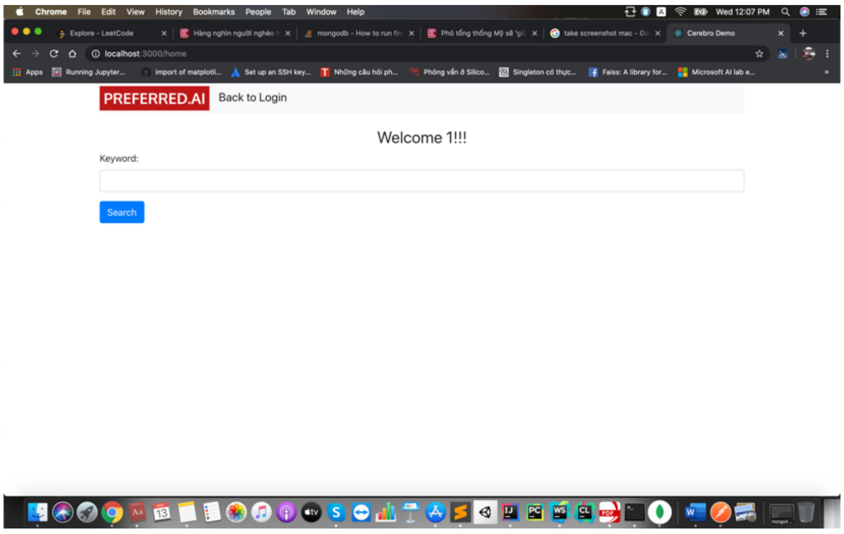
What url cerebro will make its POST request to? Go to the Cerebro project, navigate to the file **application.properties**, there you will see a line :

```
importdb.url = http://localhost:8080/blankapp/export
```

That's the url that Cerebro will a request to import its database. You will see that localhost:8080 is the host and port that your blankserver is running on. Feel free to change this url if your blankserver is running on a different host or port.

With that once the cerebro server is up and running it will automatically import your database into its dedicated database. And once that's done it will make a call to its internal Cornac to generate feature vectors for every of your users and items. Finally, build an index so that it can quickly retrieve a set of recommended items for each user.

Let’s put in the finishing touches, to our blankserver. Recall that in part 1 after finishing set up your original blankapp. If you log in this is what you see:



That’s far from an engaging UI, we would like to, for example, upon user log in show our recommendation list of items to a user.

Navigate to the **application.properties** file of the **blankserver** project. Copy this line into the file:

```
cerebro.url = http://localhost:9090
```

localhost:9090 is the default host and port that **cerebro** will run on.

Now go back to your AppController.java file and copy the function below into the class body:

```
@CrossOrigin
@RequestMapping(value="/getRecom", method = RequestMethod.POST)
public ItemListResponse getRecommendation(@Valid @RequestBody TextQuery
qObject) throws IOException,

InterruptedException,

org.json.simple.parser.ParseException {
    //object to hold request body
    JSONObject requestObj = new JSONObject();
    //we will basically forwarding this to cerebro server
    requestObj.put("text", qObject.getText());

    URL obj = new URL(cerebro_url + "/search/getRecom");
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();
    con.setRequestMethod("POST");
    con.setRequestProperty("Content-Type", "application/json; utf-8");

    // For POST only - START
    con.setDoOutput(true);
    OutputStream os = con.getOutputStream();
    os.write(requestObj.toJSONString().getBytes());
    os.flush();
    os.close();
    // For POST only - END
    int responseCode = con.getResponseCode();
    System.out.println("POST Response Code :: " + responseCode);
    ArrayList<String> ids = new ArrayList<>();
    if (responseCode == HttpURLConnection.HTTP_OK) {
        //success, parsing the resulting ids
        BufferedReader in = new BufferedReader(new InputStreamReader(
                con.getInputStream()));
        String inputLine;
        StringBuilder response = new StringBuilder();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();


        JSONParser parser = new JSONParser();
        JSONObject jsonObject = (JSONObject)
parser.parse(response.toString());
        JSONArray jsonArray = (JSONArray) jsonObject.get("ids");
        for(Object id : jsonArray){
            ids.add((String) id);
        }
    } else {
        System.out.println("POST request not worked");
    }
    //from the list of ids query the database for the actual items
    return new ItemListResponse((List<Items>)
itemsRepository.findAllById(ids));
}
```

The first two lines of this code snippet means that we are mapping a post request to the url: http://localhost:8080//blankapp/getRecom to this function. The TextQuery qObject will contain our user's id. This function is basically relaying the message back to cerebro server and get back a list of item ids. The last line mean that from the list of ids "ids" we query the database to get back the list of actual items and return back our frontend.

The last part of this will be implementing a post request to get recommendations and some GUI on our frontend to display these recommendations.

Go to the project **blankapp-frontend**. Go to src/components/homepage.js. Uncomment the code from line 7 to 20, the code at line 28, the code from line 34 to 46 and the code from line 88 to 98.

Then copy this code into line 83:

```
<h4>Personalized Recommendations:</h4>
<br/>
<table className="table table-striped">
    <thead>
    <tr>
        <th>Title</th>
        <th>Genre</th>
    </tr>
    </thead>
    <tbody>
    { this.indexList() }
    </tbody>
</table>
```

With that now restart your **blankserver** , open a terminal and navigate to the **blankserver** project and type:

```
Ctrl +C
```

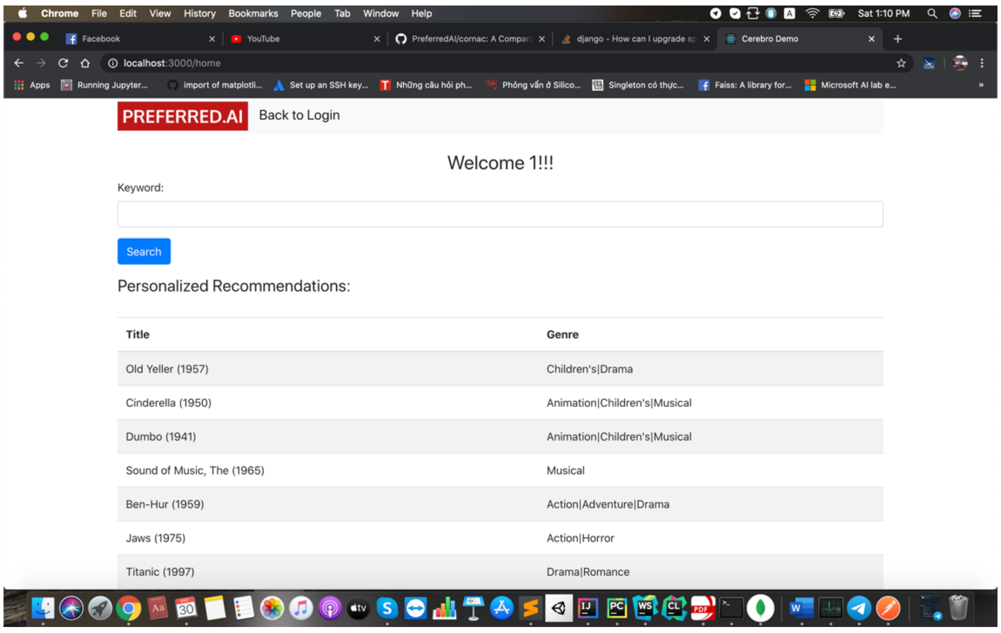
and then

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

Do the same with the cerebro project folder. This time you should wait till you see the following lines in your terminal:

```
New Index built successfully
new index searcher is ready
```

After all that you can log in again on the frontend and see that this time you are greeted by a host of movie recommendations.



Welcome 1!!!

Keyword:

Search

Personalized Recommendations:

Title	Genre
Old Yeller (1957)	Children's|Drama
Cinderella (1950)	Animation|Children's|Musical
Dumbo (1941)	Animation|Children's|Musical
Sound of Music, The (1965)	Musical
Ben-Hur (1959)	Action|Adventure|Drama
Jaws (1975)	Action|Horror
Titanic (1997)	Drama|Romance

*__Note__: you should not modify any files other than:

- In the blankserver project:
    + AppController.java
    + application.properties
- In the blankapp-frontend project:
    + homepage.js

In each of these project each files need modifying has a completed version in the "completed" package. Double check if you think you made a mistake somewhere.