Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Project Web: Deliverable 2

Armand Ciutat Camps - 48056684R
Roger Castellví Rubinat - 48056998Q
Joel Aumedes Serrano - 48051307Y
Joel Farré Cortes - 78103400T
Moises Bernaus Lechosa - 47903568L
Marc Cervera Rosell - 479880320C

Delivery date: 21-05-2021

Course 2020-21

Índex

1. Introduction

In this 2nd Deliverable we implemented our web's basic functionalities and we introduced ourselves into the world of automated web testing using Webdrivers. To test the web using the admin credentials (using the db.sqlite3 database from github) , use these:

- User: admin
- Password: 123galajat

Main changes

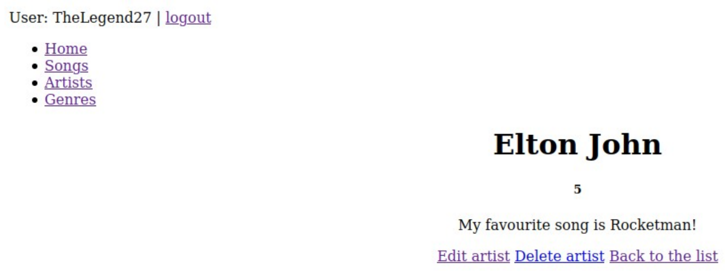
- In the Project Proposal, our application idea used Spotify's API in order to obtain the musical likings of the users for generating data. However, when starting to implement the application, it was not possible to accomplish authentication to Spotify's API, so we changed the idea to a web where the user can save songs, artists or genres by typing them.
- In "models.py" we modified the models that we used in our app. Both favourite artist and favourite genre have the same attributes: *user*, *name*, *rating*, *description*.
- The favourite song has the same attributes and the name of the artist and the genre.
- In our app folder and our templates folder we organized the project using separated folders for artists, songs and genres to keep the code small, clean and structured.

2. Design and code explanations

Creation of instances

For the creation of instances, we created a class named *CreateView*, which we will use together with a template in order to generate our website, instead of defining a function. This is called a Class Based View. This class, which is a subclass of CreateView, allows us to create a form with the instance's data. This form is created from a class we defined in *forms.py*, in which we defined said form to ask the user for all the data in the instance's model except the user, since we do not want a user to create instances for other users. With the class from the view, we defined that the user sent in the form is the same user that sent the request for the form to be posted. Once the form is completed and the object has been created, the user is shown a page with the object details created with the django class *DetailView*, along with links to modify or delete the instance or going back to the list.

Song, genres and artists use the same procedure.



User: TheLegend27 | logout

- Home
- Songs
- Artists
- Genres

Elton John

5

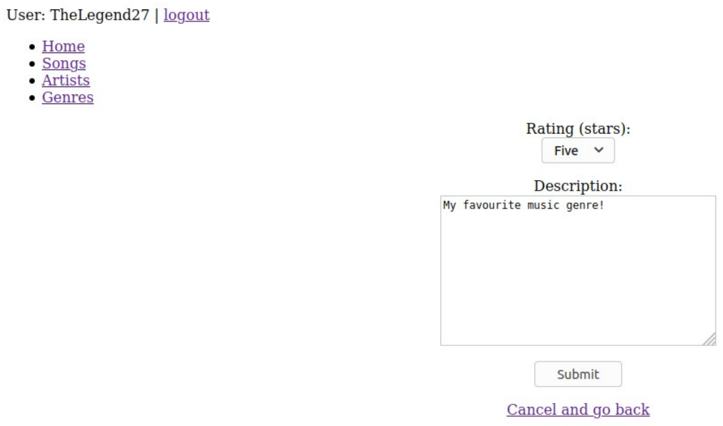
My favourite song is Rocketman!

Edit artist Delete artist Back to the list

Image 1: The page shown after adding Elton John as an artist

Modification of Instances

Once the instance has been created from its detail page, the user can enter another page to modify said instance. Using a process similar to the creation, but this time with *UpdateView*, the user now has permission to modify the rating and the description of the object, but not its name, or, in the case of the songs, neither the artist's name. Once done or cancelled, the user goes back to the details page.


User: TheLegend27 | logout

- Home
- Songs
- Artists
- Genres

Rating (stars): Five

Description: My favourite music genre!

Image 2: The page shown while modifying a genre

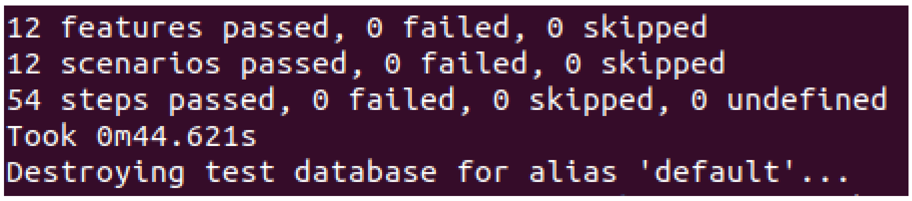
Elimination of Instances

For this case, we use a function in the views file that receives the object's id as an argument, deletes the object from the database and redirects the user back to the list of the instances. There can not be problems with the type of object to delete, because the URL we use to delete an object has the type of object in it.

Tests

To make sure our web was correctly implemented, we coded tests to check how the web behaved when a user did certain actions with it. To code these tests, we used the Selenium webdriver and the behave-django module, to implement these tests using Behaviour-Driven Development.

The tests covered all the functionalities of the web for all types of instances. For every test, we made sure that the automated webdriver made actions that would mimic a real user, and used assertions to see if the web had replied correctly to said actions.



```
12 features passed, 0 failed, 0 skipped
12 scenarios passed, 0 failed, 0 skipped
54 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m44.621s
Destroying test database for alias 'default'...
```

Image 3: Our web passes all the tests

API Use

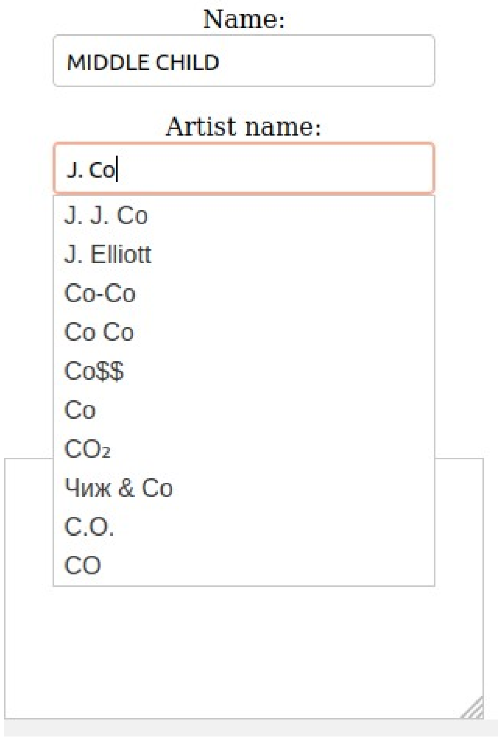
Our website used the jQuery library in order to offer the user the autocomplete function in certain fields when creating an object to the website. The function must receive a *source* parameter from which it will obtain the registries that will be offered to the user when it starts typing the desired value.

The *source* parameter can receive 3 different types of values: a JavaScript list, a String or a function. A JavaScript list does not allow us to be flexible enough and the function is too advanced for our knowledge of JavaScript. For that reason, we opted for the String option. The function assumes that the String is a URL, so that if the URL is: http://www.exemple.com, it answers with a JsonResponse with the elements to show when it receives a GET request to http://www.exemple.com?term=thing-to-search. If we pass a String with a URL to our website to the *source* parameter, and we define a view that obtains the data and returns it with a JsonResponse, we can do the request to the API in a much easier way, realizing the requests with the python *requests* module, and processing the response from the API using python instead of JavaScript.

When the user wants to create an artist, the name attribute is the one that has autocomplete, doing a request to a free API with musical data called MusicBrainz. This autocomplete does not work very well, since MusicBrainz only has one method for searching, which sorts results by exact name match before the popularity of registries that start with the search term. So if the user types “Que” waiting for “Queen” to pop up, before it will get lesser known bands with names like Que, [.que], or Gee-Que since they are more similar to the search term.

When the user wants to create a genre, the name attribute does the autocomplete, reading from a file which contains approximately 1800 different genres.

When the user wants to create a song, autocomplete is made in 2 attributes: genre and artist name. The genre attribute provides the same data as if a genre is created, reading from the file. On the other hand, while autocompleting the artist name, the field will first show the users the artists that they have added to the web (as long as they match with the search term), and then, if less than 10 artists have been shown to the user, the function does a search to the MusicBrainz API, adding the necessary artists to the end of the response in order to give priority to the saved artists.



Name:

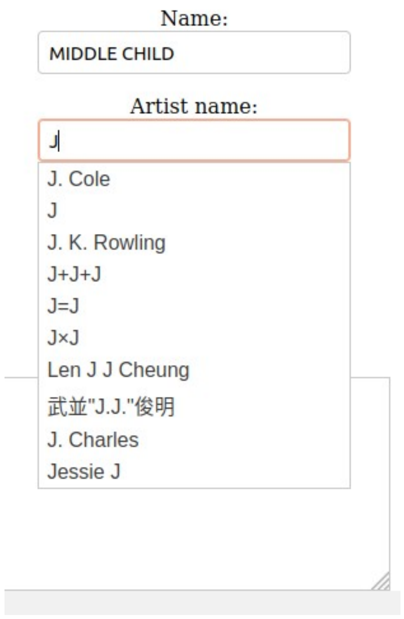
MIDDLE CHILD

Artist name:

J. Co

J. J. Co
J. Elliott
Co-Co
Co Co
Co$$
Co
CO₂
Чиж & Co
C.O.
CO

Image 4: Trying to add "MIDDLE CHILD" by "J. Cole" without having "J. Cole" as an artist



Name:

MIDDLE CHILD

Artist name:

J

J. Cole
J
J. K. Rowling
J+J+J
J=J
J×J
Len J J Cheung
武並"J.J."俊明
J. Charles
Jessie J

Image 5: Trying to add "MIDDLE CHILD" by "J. Cole" after adding "J. Cole" as an artist