Limelighter Documentation

> Limelighter is a social media for musicians and people that want to work with them. The app facilitates the exposure of upcoming artists, keeps the data of all this groups and individuals for all to see, makes communication easier, makes administrating a group way more managable, between other things, all in one package.

 Features

- Share links for your favorite pages in your profile
- Share videos from multiple media services and include Spotify tracks, albums and artists in your profile
- Add an instrument in your profile, and thus become a musician
- As a musician you can:
    - Create and apply to Groups
- Create and apply to Announcemnts
- Chat with group members and other people in the app
- Rate a user based on his announcements
- Rate a group where you are a member

Arquitecture

The application uses an MVC arquitecture with an object relation mapping for connecting the entities in the database with objects in the app (ORM).

 Primary Entities

Entity	Description	Attributes
User	Represents the user that navigates in the application and saves it's atributtes	`first_name:string` `last_name:string` `email:string` `password:string` `country:string` `city:string` `state_region_province:string` `adress:string` `phone_number:string` `rating:float` `musician:boolean` `instruments:string[]` `links:string[]` `allratings:integer` `confirmed:boolean` `photo:text`
Group	Represents a group of users with atributte `musician: true` and saves it's atributtes	`name:string` `date:date` `instruments:string[]` `genre:string` `description:text` `avaincies:integer` `leader_id:integer` `rating:float` `postulants:integer[]` `allratings:integer` `photo:text`

Entity	Description	Attributes
Announcement	Represents an announcement made by either `user` or `group` and saves it's attributes	`title:string` `description:text` `date:date` `userId:integer` `groupId:integer` `postulants:integer[]`
Postulation	Represents an application for an `announcement` or `group` made by a `user` and saves it's attributes	`description:text` `date:date` `status:string` `userId:integer` `announcementId:integer` `groupId:integer`

- User
  - **musician**: defines if the user has or not become a musician by adding an instrument in his profile
  - **instruments**: it's an array of the instruments played by the user
  - **links**: it's an array of the user's media
  - **allratings**: stores the ammount of times that a user has been rated
  - **confirmed**: attributes that tells if the user has confirmed his email
  - **photo**: url of the photo in the storage system cloudinary
- Group
  - **instruments**: it's an array of the instruments played in that group
  - **avaincies**: represents the maximum ammount of members that a group can have.
  - **postulants**: it's an array of user's id that had postulated to the group, making that users have an unique postulation to the group
- Announcement
  - **postulants**: it's an array of user's id that had postulated to the announcement, making that users have an unique postulation to the annoucement
- Postulation
  - **status**: status of the postulation it has 3 possible states: accepted, pending and rejected.

👶🏽 Secondary Entities

Entity	Description	Attributes
Chatmessage	Stores a chat message for a certain group via the `groupId` attribute	`content:string` `userName:string` `userId:integer` `groupId:integer` `date:date`
Groupvideo	Stores the url of a video for a certain group via the `groupId` attribute	`groupId:integer` `title:string` `videoUrl:text`

Entity	Description	Attributes
Spotifymediagroup	Stores the URI for a Spotify asset for a certain group via the `groupId` attribute	`groupId:integer uri:string type:string`
Spotifymediauser	Stores the URI for a Spotify asset for a certain user via the `userId` attribute	`userId:integer uri:string type:string`
Userannouncement	Represents an assoication between a user and an announcement (1:N).	`userId:integer announcementId:integer rated:boolean`
Userchatmessage	Stores a chat message for a conversation between two users via the `userId` and `chatId` attributes	`content:string userName:string userId:integer chatId:integer date:date`
Usergroup	Represents an assoication between a user and a group (N:N).	`userId:integer groupId:integer rated:boolean instrument:string`
Uservideo	Stores the url of a video for a certain user via the `userId` attribute	`userId:integer title:string videoUrl:text`

- Userannouncement
  - **rated**: defines if a user has or not been rated by another user who applied to an announcement.
- Usergroup
  - **rated**: defines if a group has or not been rated by a user who applied to a that group.
  - **instrument**: defines an association between a user and an instument in a group.

Layout

The page uses scss for the style, and the structure of the page can be summarized with: a sidebar and the content. The sidebar makes navigability a much easier and intuitive task. It also helps to show if the user is a musician or not, wich is done with condicionals and different classes in css. The groups and announcements are shown in a card form, and this cards are shown in a 3x3 grid (made possible by the css display "grid"). The users and postulations are shown as rows, made possible with a flexbox display in css.

Code

Flow

A new user can choose to view the app anonimously or to create an account. If he doesn't create an account, he can view groups, announcements and users, withouy being able to apply to any of them or create a resource. If he does create an account, he has to enter his full name, email and password. He can enter more information about himself, but these are optional and can be added later. Once logged in, he starts off as a non musician user, meaning that he can only create announcements and apply to these, but can only see groups. He can also chat with another user and add links and media in his own profile. In here, he can also add an instrument of his liking and once this is done, he changes automatically his status to musician. Also, being

logged in, two more options appear in the sidebar: one to view an manage his application and another one to view his profile. Once he is a musician he can create groups and apply to them, also being able to chat with the other members of the group. As a group leader, he can accept or reject applications to his group, edit the group parameters and expel any member from the group. Also, as a group leader he is also able to add media (videos and Spotify assets) to the group profile and delete them at will. In the profile section, a user is able to edit his profile and delete his account if he wants. If this user was leader of any groups, the leader position to that group is transferred to the latest user to have joined. If he wasn't a group leader, he just leaves any group he belonged to automatically.

Router

Announcements

There are eight methods in this router.

- **announcements.list**: shows all the announcements there are, separating between announcements belonging to that user and announcements that does not belong to that user. It then renders a view that displays all these announcements as cards.
- **announcements.new**: renders a view that shows a form to create an announcement. Then **announcements.create** recieves this request and creates an instance of an announcement in the database, redirecting then to the list of announcements.
- **announcements.edit**: gets an announcement on a certain id parameter, and renders a view loading this annoucement into a form. Once submitted, this request is recieved by the **announcements.update** method, which updates the instance of that announcement in the database.
- **announcements.delete**: deletes an announcement from the database and redirects to the list of announcements.
- **announcements.show**: gets an announcement by an id parameter and then renders a view that shows this announcement in detail, along with all the applications to this announcement.
- **announcements.rate**: patch method that recieves a request with a rating for the creator of the announcement. Redirect to announcement once the database is updated with the rating.

Announcement_Postulations

- **announcement_postulations.new**: renders a view for a form to create a postulation for an announcement then **announcement_postulations.create**: recieves this request and creates an instance of a postulation in the database, redirecting then to the respective announcement. It also sends an email to the owner of the announcement which is being applied to (with the information of the postulation).
- **announcement_postulations.accept**: changes the status of a given postulation (of an announcement), from 'Pending' to 'Accepted'. It also sends an email to the owner of the postulation saying that the postulation has been accepted.
- **announcement_postulations.reject**: changes the status of a given postulation (of an announcement), from 'Pending' to 'Accpeted'. It also sends an email to the owner of the postulation saying that the postulation has been rejected.
- **announcement_postulations.delete**: deletes a postulation from the database and redirects to the respective announcement.

Group_Postulations

- **group_postulations.new**: renders a view for a form to create a postulation for a group then **group_postulations.create**: recieves this request and creates an instance of a postulation in the database, redirecting then to the respective group. It also sends an email to the owner of the group which is being applied to (with the information of the postulation).
- **group_postulations.accept**: changes the status of a given postulation (of a group), from 'Pending' to 'Accepted'. It also sends an email to the owner of the postulation saying that the postulation has been accepted.
- **group_postulations.reject**: changes the status of a given postulation (of an group), from 'Pending' to 'Accpeted'. It also sends an email to the owner of the postulation saying that the postulation has been rejected.
- **group_postulations.delete**: deletes a postulation from the database and redirects to the respective group.

Group_Media

- **group_media.show**: gets a group given an id, from it, gets all its videos and Spotify assets, rendering them in a view that renders a react component which shows a form that gives all the Spotify players and recieves a URI as an input.
- **group_media.add**: given the request from the "show" method, creates a groupvideo instance and adds it to the database; then it redirects to the **group_media.show** with the same id of the group.
- **group_media.addSpotify**: gets a group given an id and given the request in the "show" method, it builds a spotifymediagroup instance, saves it in the database and redirects to the **group_media.show** with the same id of the group.
- **group_media.delete**: the react components that render the media have an integrated delete button which submits a form that sends a request to this method, deleting the given type of media.

Groups

There are eleven methods in this router:

- **groups.list**: shows all the groups there are, separating between groups that the user belongs to and groups that the user does not belong to. It then renders a view that displays all these groups as cards.
- **groups.new**: renders a view that shows a react component. This component renders a form to create a group. (One of the fields imports data to show all music genres in a select tag). Then **groups.create** recieves this request and creates an instance of a group in the database, redirecting then to the list of groups.
- **groups.edit**: gets a group on a certain id parameter, and renders a view loading this group into a react component that renders a form. Once submitted, this request is recieved by the **groups.update** method, which updates the instance of that group in the database.
- **groups.delete**: deletes a group from the database and redirects to the list of groups.
- **groups.show**: gets a group by an id parameter and then renders a view that shows this group in detail, along with all the applications to this group, all the members of the group and all the available (and wanted) instruments for that group.
- **groups.rate**: patch method that recieves a request with a rating for the group. Redirect to announcement once the database is updated with the rating. The rating comes from a react component that lets the user submit a rating after 4 days of being accepted into the group. Then, once it saves the reference that the user already rated the group, it redirects tho the **groups.show**.

- **groups.chat**: gets all the chat messages of a group given the id of the group, then renders a view of a chatbox which establishes a connection between the group members using sockets. (The code for the sockets is located in src/assets/js/code/group_chat_connection.js and in src/app.js).
- **groups.saveMessage**: recieves the request from the group chat, which is an array with the messages sent by the user. Once is saves all the messages, it redirects to the **groups.show**.

Index

There are two methods

- **/(root)**: renders the discover of the application, which shows some of the recent groups and announcements
- **confirmation/:token**: gets a confirmation using a unique token and updates the attribute confirmed from the user corresponding to the id related with the token.

Postulations

There are three methods in this router:

- **postulations.list**: this method is used only to present the API.
- **postulations.user**: shows all the postulations that belong to the user using the userId as parameter.
- **postulations.delete**: deletes a postulation using the given id as parameter.

Searches

All searches work in the same way, they get what has been searched and the selected search criteria. The search criteria is what we use in a switch/case block, so that we know how to use what has been searched. This switch/case block is what we use to filter the entities that we will show to the current user. It is also useful to notice that this methods are all posts, they work with the get method that lists the respective entity.

- **searches.users**: The search for users is the simplest. We find all the users that are not the current user in the database (if the user is not logged in, we do not filter anything). Then we filter all the users. In the case of 'email', 'country', and 'city' we just see if what was searched is included in the respective attribute of the user. For 'instruments' it is the same, but we make this operation for all the instruments in the array of instruments that the user has, or until it finds an instrument that meets this condition.
- **searches.groups**: Here it is needed to make a distinction between the groups that the user is part of and which are not. After that we make a list with all the leaders and occupants, which we need in the view of the page to show the data correctly. In the filtering step, we need to get the group leader of each group to then use it in the case that the search criteria is 'leader', this is why it is also useful to have the leader list before the filtering. The other search criteria that is new is 'vacancies', which is for minimun vacancies, and we filter this with just a comparison:`return group.avaincies - occupantsList[index] >= searched;`. After we have filtered the groups, we have to make the leader list and occupant list again, because we now have less elements in this list, and this because we have less groups to show.
- **searches.announcements**: This is really similar to the groups search. We apply the same idea of a leader list, but with the author of the announcement.

Session

There are three methods in this router

- **session.new**: renders a view that shows a form to logging into the aplication
- **session.create**: recieves the request from session.new and verify if the email and passwords are correct, if they are it redirects to the root if not it redirects to the session.new with the corresponding error.
- **session.destroy**: it works as a logout destroying the current session and redirecting to session.new

Users_Media

- **users_media.show**: gets a user given an id, from it, gets all its videos and Spotify assets, rendering them in a view that renders a react component which shows a form that gives all the Spotify players and recieves a URI as an input.
- **user_media.add**: given the request from the "show" method, creates a uservideo instance and adds it to the database; then it redirects to the **user_media.show** with the same id of the user.
- **user_media.addSpotify**: gets a user given an id and given the request in the "show" method, it builds a spotifymediauser instance, saves it in the database and redirects to the **user_media.show** with the same id of the user.
- **group_media.delete**: the react components that render the media have an integrated delete button which submits a form that sends a request to this method, deleting the given type of media.

Users

There are eleven methods in this router:

- **users.new**: renders a view for a form to create a user.
- **users.create**: recieves the request from the new user view and creates an instance of a user in the database. It also sends a unique link, generated with token, to the user email so he can confirm his registration and start using the aplication. This method also handles uploading images to the storage system Cloudinary.
- **users.list**:shows all the users there are, separating between. It then renders a view that displays all these users as cards.
- **users.edit**: gets a user on a certain id parameter, and renders a view loading this group into a form.
- **users.update**: it updates the instance of user in the database and it can also manage the update of the profile photo in Cloudinary.
- **users.delete**: deletes a user from the database and it's corresponding profile photo on Cloudinary. Due to foreign keys it also deletes all the announcements, postulations, chatmessages, spotifymediausers, userannouncement, userchatmessage, usergroup and uservideos that belongs to the user. In the case of groups that belong to the user, meaning he is the leader, if the group is only compound by him, it will be deleted if there are other users in it, it will transfer the leadership to another user.
- **users.show**: gets a user by an id parameter and then renders a view that shows this user in detail, along with all his instruments and links.
- **users.addInstrumentsAndLinks**: it recieves the request of the users.show method and updates the instruments and the links of the user in the database.
- **users.removeLink**: it deletes the link from the array of links of the user and updates the array in the database.
- **users.chat**: creates a chatroom with another user, with both ids, since they are unique the chatroom in unique. It then renders this chatroom

- **users.saveMessage**: saves the messages from the user that were sent to the corresponding chatroom.

React

app.js

Imports all the react components and renders them if they are being shown in a view.

AcceptPostulation

Renders a pop up that lets the leader of the group assign an instrument to the user that is being accepted. The objetive is to give the option of accepting a user based on the instruments he plays.

AddMedia

Renders a form that allows the user to add any type of graphic media (Youtube, Twitch, Soundcloud, etc.), giving the title and the url. A function checks the url on change, not allowing any domain that is not registered in src/exports/safe_urls.js.

AnouncementRate

Renders a form with 5 radio buttons represented by an svg that show stars. The form paints the stars given the onMouseEnter and onMouseLeave functions, allowing for a better UI when rating a user. The constructor also defines a variable called timeSpan, which represents the time in days that the user has to wait to be able to submit a rate. Once this time has passed, the form is "unlocked".

GroupRate

Renders a form with 5 radio buttons represented by an svg that show stars. The form paints the stars given the onMouseEnter and onMouseLeave functions, allowing for a better UI when rating a group. The constructor also defines a variable called timeSpan, which represents the time in days that the user has to wait to be able to submit a rate. Once this time has passed, the form is "unlocked".

ShowMedia

Renders a ReactPlayer that given a url shows a video that the user has added. Every video also has a form that represents a delete button (with MdClose) to delete a video.

SidebarIcons

Renders the sidebar with various icons next to anchors that lead to each page. It checks if the user is logged or if he is a musician and given this values, it renders diferent types of sidebars.

SpotifyMedia

Recieves json objects that contain the Spotify albums, artists and tracks of a user in form of arrays, then parses those objects and renders a ReactPlayer given the URI for each one of the elements in those arrays.

Stars

Given the rating and the amount of ratings, Renders five svg stars, painting yellow the ones that correspond to the rating and painting gray the ones that do not.