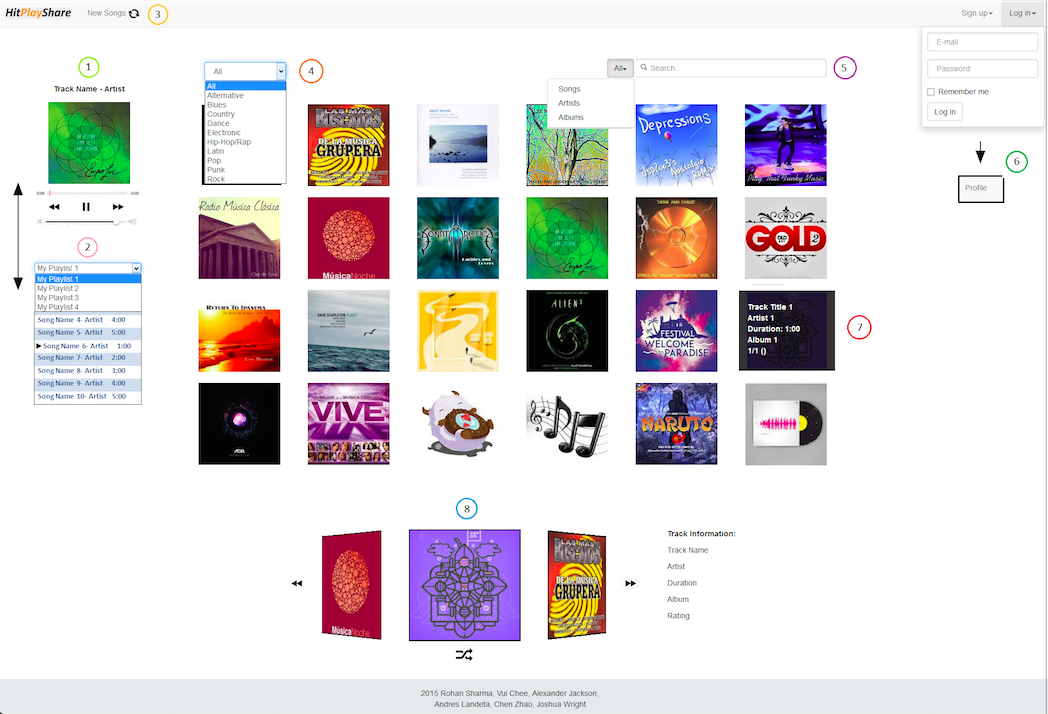
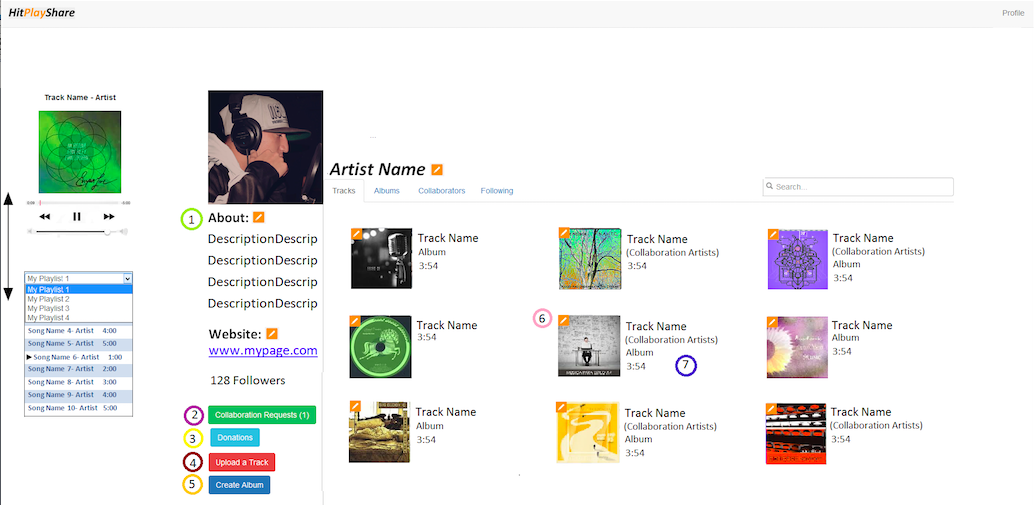
***Part C : Project Plan and Application Design***

***User Interface Design***

***Home Page***

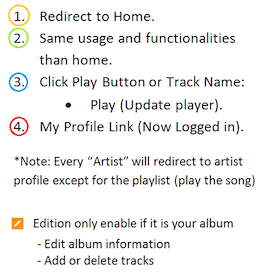


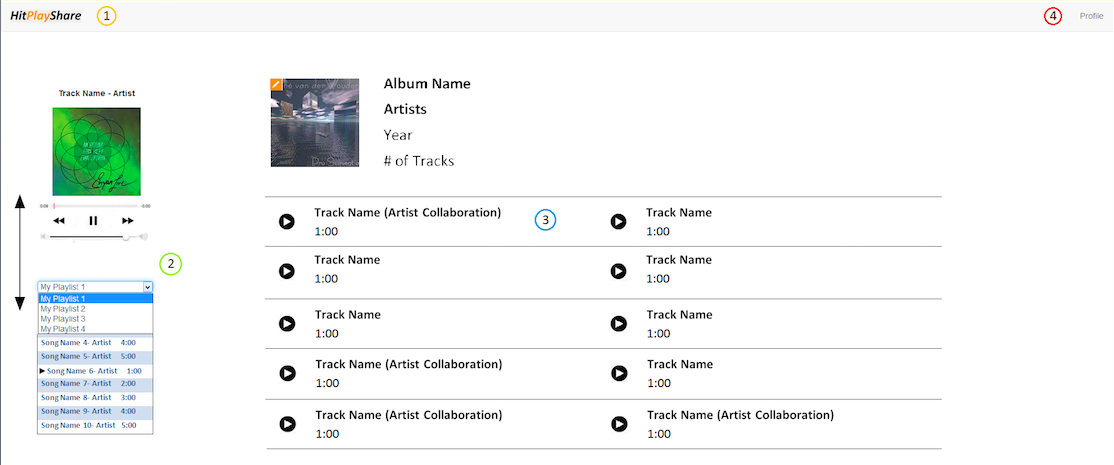
***User Profile Page***

******

***Album Page***

***Album Page***



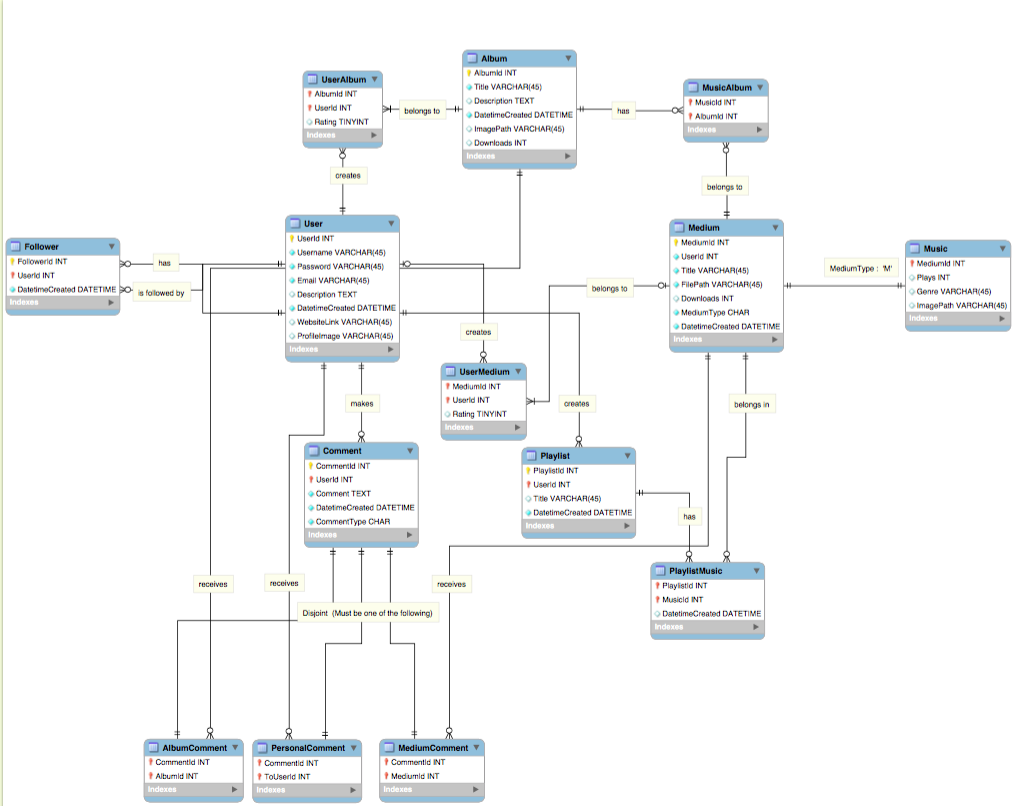


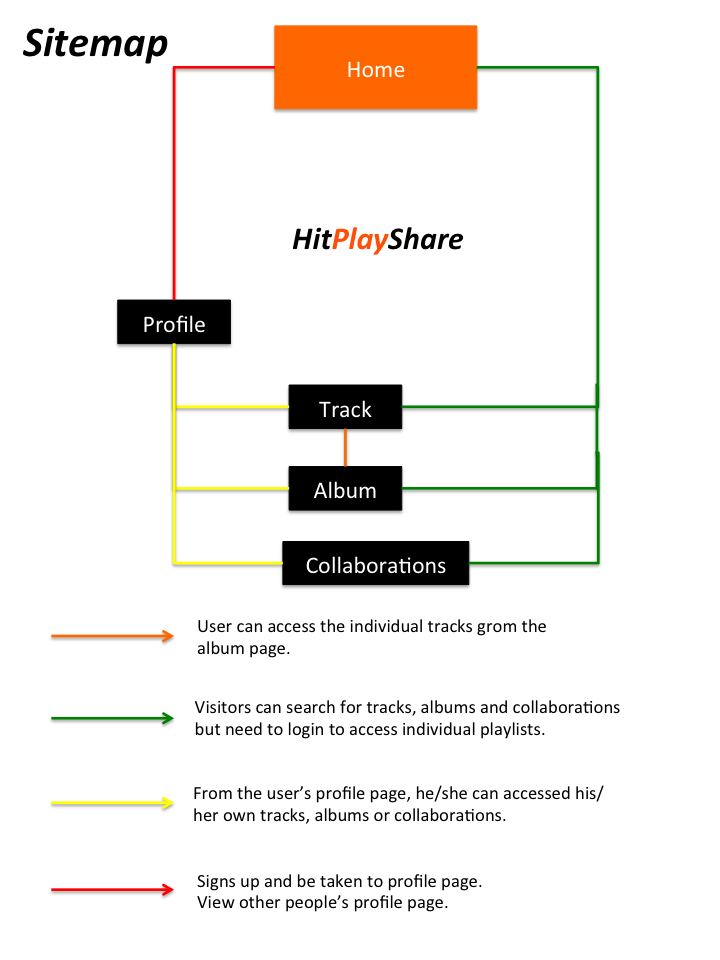
***User Profile Page***

1. Inline Edition of artist name, description and website
2. Opens a window with the collaboration requests received (accept/decline)
3. Let you set up your donations
4. Opens a window where you can upload a track and set the information
5. Opens a window where you select your album name and confirm the creation:
   1. Upon confirming, a new browser tab is opened where you can edit the album
6. A new window for:
   1. Change name
   2. Edit collaborators
   3. Edit album linked to track
   4. Delete track/album
   5. Following tab:
      1. Allows you to delete artist you follow
7. Click track tab:
   1. Cover or name: play track
   2. Album: change to album tab, or search for album
   3. Collaboration: change tab or search for artist
8. Click album tab:
   1. Cover/name: redirect to album page
   2. Collaboration artists: change tab or search for artist

***Home Page***

1. Current track playing with sound control
2. Playlist Menu:
   1. Display when the user is logged in
   2. Position will go down with the page
   3. Change what the playlist will be displayed
   4. Current track playing is highlighted
   5. Tracks order can be modified
   6. Tracks can be deleted from playlist
3. Reload new tracks for the grind
4. Select track’s genre (will update the grind)
5. Search bar:
   1. Search for tracks, artists, albums or all of them
   2. Will update the grind with results
6. Log in form will change to link to profile
7. Grind element:
   1. Hover:
      1. Show information
   2. Click:
      1. Track -> play and update the carousel
      2. Artist -> redirect to profile
      3. Album -> Redirect to page
8. Carousel with Random Tracks:
   1. A track will be added when clicked on grind
   2. You can return to previous tracks that you played
   3. Generate a random track to play
   4. Information on the right will update

***Data Design***



***Navigation Design***

***Technology and Infrastructure***

As previously shown, we have used a sitemap and a basic user interface design to demonstrate basic workings of our planned website. Now, we are going to talk about the Rails framework and architecture we will be using in construction of our web application. In addition to using Ruby on Rails, we will be using JavaScript (jQuery and plug-ins) and CSS for designing the eye-pleasing pages as well as built-in interactivity to our pages. Moreover, most user activity will stay on the same page thanks to Ajax.

First off, I am going to give an overview of a Rails application when a user types “**rails new app\_name**” and creates a new Rails application. This will generate a bunch of files in a directory under the specified application name.

***Overview of File Structure of Rails Application***

App\_name (main directory/ ones bolded indicated most important aspects)

1. **app (contains assets folder -> stylesheets, JavaScript, images)**
2. bin
3. **config (for the report, only routes.db is discussed)**
4. **db (migrations)**
5. lib
6. log
7. public
8. test (for testing models)
9. tmp
10. vendor
11. other files (Gems etc. )

Under the app folder inside your application directory, there are 6 folders: assets, controllers, helpers, mailers, models and views. And in this folder, most of your code is written here. For front–end code such as Javascript and CSS, they can be written separately and stored inside the assets directory. For ruby code, most of the code will be written in models, controllers and views. Obviously, the Rails framework adopts the model-view-controller (MVC) approach by allowing users to separate tasks under this 3 broad categories. Another important folder is the config folder which stores files that allows users to route different pages together using ruby code.

***Database Management***

For database management, like performing CRUD on the selected database, this is done primarily through the db folder, which houses the migrations folder. In Rails, actions on the database can be performed using “migrations”. For instance, if the user wants to add a column to a table, he can alter the migration file and call “rake db:migrate” to update the database with the new migration file. Through migrations, the user can avoid having to write nasty SQL statements to change or modify certain parts of the database.

One particular thing about rails, if you want to perform queries on the database, you can simply call “rails console” on your command prompt and it will give you a console where you can query the database using object oriented language. So if you want to check a record in the User table, you can just type : “User.find(1)” , to find a record where the user id is 1.

***Models***

Another important thing about rails is that tables are called models (ruby classes). To create a model in Rails, the user can call “rails g model **model\_name** [field\_name:field\_type]” to construct a class in the models folder and a migration file which specifies how the table should be created in the database. As mentioned previously, any desired modification to the table can be done through the migration file.

For relationships between entities in database modeling, they are done through the models in Rails, where special ruby syntax is used to specify many-to-many or other types of relationships.

To view all completed models after migrating, the db folder stores the schema.rb file which displays all ruby classes that were generated so that the user can keep track of all migrations regarding built tables.

***Views and Controllers***

The most important part of the Rails application is the controller folder, which contains the controller files which interacts with models and views of the application.

To understand how the controller works, we must also understand how html files are stored in Rails. In a Rails application, html code are stored in separate folders under a **views** folders. The folders named here are related in name to ruby files stored in the **controller** folder.

Eg. A controller file named media\_controller.rb has all the actions required by all html pages stored in the media folder (under views folder).

**Controllers**

* **media\_controller.rb**

**Views**

* **media**
  + **index.html.erb**
  + **new.html.erb**

Each html form inside Rails must correspond to a method inside the controller. So for every file inside the media folder, the prefix matches index and new methods located inside media\_controller.rb.

So how do they interact? For example, if you want to submit a form from a html page, the values from the form are passed to the controller method (called an action in Rails) specified by the route inside routes.rb in the config folder. The method processes the values from the form and redirects the user accordingly. Since it is a class method, it is up to the creator to modify what it does.

When a controller needs to interact with a model, a method inside the controller file creates an object of that model class which is inside the models folder. Models basically serve as classes which are used inside controllers.

For html.erb files that requires JavaScript files, the <%= javascript\_include\_tag “some\_file.js” %> tag includes the appropriate JavaScript file from assets directory without needing the user to specify its path. Similarly for CSS files, the <%= stylesheet\_link\_tag “some\_file.css” %> does the same.

The clever use of routes and controller methods allows information to be passed from model to controller and to views and vice-versa.

***Security Design***

***Technology Based Risks and Design Choices***

***Risks associated with Rails***

**SQL Injection**

Ruby on Rails is often used with an ORM ( object-relational mapping ) called ActiveRecord. Rails applications often use methods on models to query data, luckily in most cases, SQL injection is prevented as part of Rails. However, it is still possible to conduct an SQL injection using Ruby on Rails code.

**Sessions**

By default, Ruby on Rails used a cookie based session store. What that means is unless you change something,, the session will not expire on the server. That also means that some default applications are vulnerable to replay attacks. Sensitive data should not be stored in the session.

**Authentication Attacks**

Rails does not provide authentication itself. Web developers have to use Rails leverage libraries such as Devise to provide authentication. Hence, the risk of authentication attacks falls under the libraries which help authenticate information between servers.

**Insecure Direct Object Referencing**

Rails apps use a RESTful uri structure. This means that paths between files are intuitive and guessable. As a result, other users can access and modify data belonging to another user. Default Rails applications have no protection against this threat.

**Cross-site Scripting (XSS)**

**Sometimes, during the development of Rails front-end html pages, developers might like to pass Ruby class variables using <% raw @user.name %> tags in .erb files. The problem with this is when the servers receive html content from users which could be malicious.**

**Sensitive Files**

**One thing to note about Ruby on Rails applications is that open source apps hosted publicly with available code repositories. This implies that some files containing sensitive information may be infiltrated by outsiders if not carefully managed.**

**YAML Deserialization**

**Ruby on Rails uses a serialization format called YAML (simple language that describe data). The main issue is Rails sometimes de-serialize certain objects of classes. What this results in is people can initialize objects from well-chosen classes and cause arbitrary code to be executed, without requiring any permissions from the victim application.**

***Access Control***

User management and authorization are not core features in Rails. Which implies that access controls of different users have to be implemented using other means instead. In Rails, fortunately, we can download “gems” to provide different forms of authentication in different parts of the application, such as signing in as a user with other accounts in Twitter or Facebook. However, authentication is not authorization itself. We need to somehow incorporate limits to what users can see or do.

Most web applications employ the role-based authorization. This means that users conforming to certain roles can access only parts of the website that they are entitled to. This is easy to implement simply by distributing attributes of roles to different users. As such, those who possesses an incorrect role will be denied access from parts of the application.

***Rails Interoperability***

Since Rails runs on Ruby code, and to compile ruby code one just needs to install a Ruby interpreter, therefore Ruby on Rails can be run on any operating system so long as the interpreter is present and Rails is installed. Furthermore, Rails operate based on MVC model, which means there is separation among structure, presentation and behavior in web pages, of which is an element of interoperability. Modifying Rails applications to include functionality on mobile devices is also possible since Rails is a cross-platform framework.

***Risk Management Plan***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Risk | Trigger | Likelihood | | Impact | Contingencies |
| SQL injection | Database is changed arbitrarily | Likely (Rails has guards against this though) | | **(Major)**   1. User information may be stolen or modified 2. Modified database tables may interfere with applications operability | Do not build SQL statements based on user-input |
| Sessions | Confidentiality attacks are hard to detect | Possible | | **(Minor)**  Sensitive data such as passwords may be stolen if stored in sessions | Do not store sensitive information in sessions |
| Authentication attacks | Server side receives requests from unknown parties | Possible | | **(Moderate)**  May accidentally distribute information to parties unintended for | Secure session strings can be used to ward off strangers |
| Insecure Direct Object Referencing | Since models are linked to tables in the database, outsiders initializing objects may include records that do not belong to the database | Likely | | **(Moderate)**  Records that do not belong to the database may be interfere with queries of other records, resulting in incorrect information being sent back to the user | Use resource-base access control libraries such as “cancancan” to allow authorization of database operations on objects by business logic of application |
| Cross-Scripting Attacks | One potential sign is that users may experience unintended pop-up scripts when there is none | Likely | | **(Major)**  One major threat is that malicious scripts can access cookies, sessions or other sensitive information retained by browser | Do not pass ruby variables using the output tag or accept unverified data from unknown sources |
| Sensitive Files | Confidentiality attacks here are common | Likely | **(Major)**   1. Outsiders may steal secrets to hash cookies or other sensitive information 2. If important application secrets are exposed, people can completely disrupt the web application | | Exclude or carefully manage publicly available files containing such information |
| YAML De-serialization | One can test this concept on application to see if it is vulnerable to this type of risk | Likely | **(Major)**  Strangers can execute ruby code at the privilege level, thereby leading to host takeover | | Consistently update Rails version to include all security updates |

***Testing Plan***

For our project there will be two areas of testing that we will be focusing on. The first will be unit testing to ensure that our software functions correctly and that edge cases and security risks will be accounted for. Unit testing will be written for larger more complicated parts of the project, particularly the database interface and the various Rails controllers. The second realm of testing will be usability testing. This will focus on developing an intuitive and clean user interface catered to the user's needs and problems, while keeping the overall goal of the project in perspective (a music streaming site with a focus on promoting new and little known artists).

**Unit Testing**

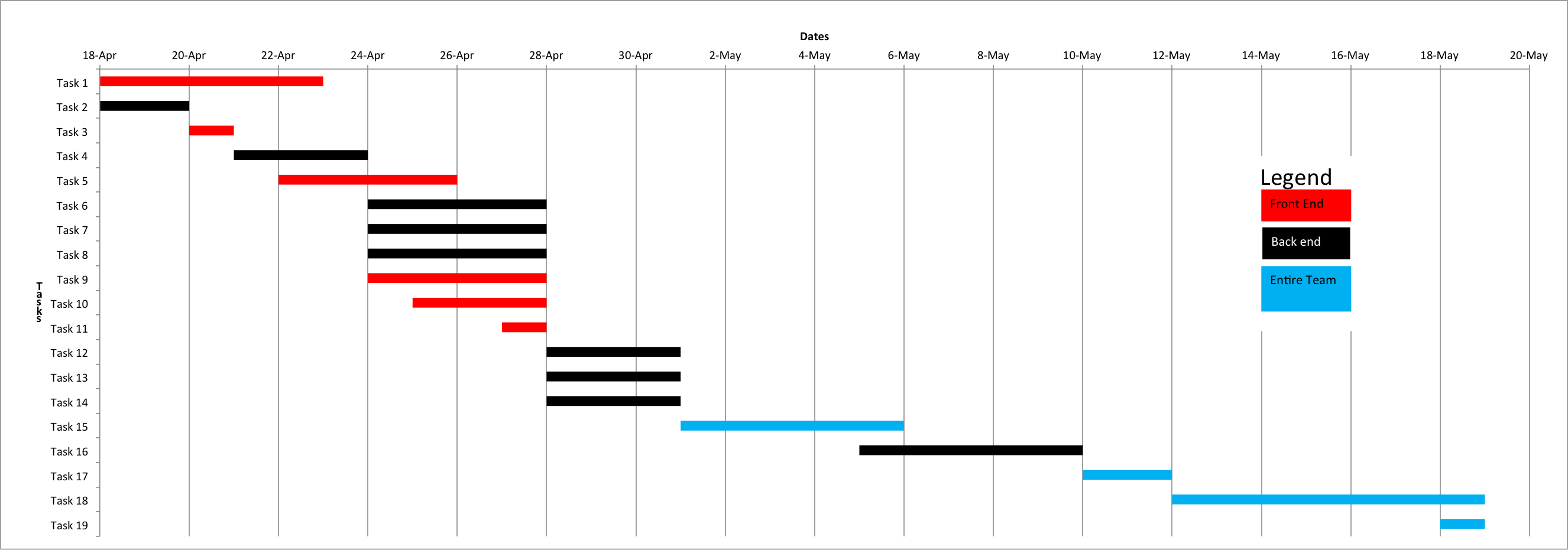
For the backend, unit testing will ensure that the various pieces of the project work. By building a unit test that plugs directly into the interface of a controller, database interface, etc. we will be encouraging both working and well tested code as well as loose coupling between parts of the backend. This will also help to avoid issues with code not quite working correctly after minor refactors. If any code is changed in a file with unit testing, the unit tests will need to be run successfully before the code is committed and pushed to the master repository. Unit testing will also allow the group to agree upon how specifically each part of the project will work. Unit tests will be written before an important file or module is implemented, so that team members will know both how that portion of code will work as well as the correct way to interface other parts with it. Both white and black box testing will be used in this process, white box to ensure that the various modules of the project are working properly, and black box to ensure that the various modules of the project interface together properly and seamlessly. By both white and black box testing, we will further promoting loose coupling and modularity in the project. \*SAY WHICH PIECES OF THE PROJECT WILL USE UNIT TESTING\* Unit tests will be written using Ruby on Rails' built in testing framework.

**Usability Testing**

We plan to do frequent user testing and filter user's feedback when using the site back into the development of the product itself. Once an initial working prototype is created, we will begin this process by regularly having actual users (with no knowledge of the workings of the project) attempt to complete various tasks integral to the sites functionality. These might include uploading a song, playing a song, searching for a user, etc. By doing this, we will both ensure that our interface is intuitive enough for users to use it.

Other forms of usability testing will include asking a potential user with a vested interest in the product, such as an amateur musician, to use the product and without giving them any help observe how they interact and work with the program. Although this process would be more time consuming than simply asking them to do certain tasks, it will allow us to see how users use the site, and what sorts of changes are necessary for us to make, in order to make apparent all the features and functionalities of the site. Another benefit of this process is that it will allow us to see what sorts of features the users find useful and what they do not. Although the benefits of this are limited given a development phase of only a few weeks, the process could still allow us to see major flaws in our design or perhaps improve on certain features that users find especially attractive. If the project was both longer and larger scale we might implement A/B testing during the development of specific features, but given the small timeframe and scale A/B testing is unfeasible.

***Project Timeline Chart***



***Project Timeline***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Description** | **Start Date** | **End Date** | **Duration (Days)** | **Extended Description** | **Responsibility** |
| Task 1 | 18/4/2015 | 22/4/2015 | 5 | Complete UI design for the home page. | Front End team |
| Task 2 | 18/4/2015 | 19/4/2015 | 2 | Allow user to sign up in Rails application. | Rohan |
| Task 3 | 20/4/2015 | 20/4/2015 | 1 | Test the user sign up. | Front End team |
| Task 4 | 21/4/2015 | 23/4/2015 | 3 | Allow user to sign in the Rails application (Sessions). | Rohan |
| Task 5 | 22/4/2015 | 25/4/2015 | 4 | Complete UI design for the profile page | Front End team |
| Task 6 | 24/4/2015 | 27/4/2015 | 4 | Allow music to display algorithmically and dynamically on the front page. | Rohan |
| Task 7 | 24/4/2015 | 27/4/2015 | 4 | Allow users to maintain their profile, follow other users and view other profiles and show music on profile. | Vui Chee |
| Task 8 | 24/4/2015 | 27/4/2015 | 4 | Create forms which will be used by the User to upload images/music with information. | Alex |
| Task 9 | 24/4/2015 | 27/4/2015 | 4 | Work on implementing an instance of jplayer (audio player) which will play music on the homepage. | Josh |
| Task 10 | 25/4/2015 | 27/4/2015 | 3 | Complete UI design when you click music/album + comments | Front End team |
| Task 11 | 27/4/2015 | 27/4/2015 | 1 | Test all the functionalities implemented form Task 3 to Task 7. | Front End team |
| Task 12 | 28/4/2015 | 30/4/2015 | 3 | Provide the functionality for users to rate music/download music +Donate to artists | Rohan |
| Task 13 | 28/4/2015 | 30/4/2015 | 3 | Provide the functionality for users to comment on profiles/music and respond to personal messages. | Vui Chee |
| Task 14 | 28/4/2015 | 30/4/2015 | 3 | Implement search functionality for users to search artists/music/profiles | Alex |
| Task 15 | 5/1/15 | 5/5/15 | 5 | Allow users to maintain(CRUD) multiple playlist using cookies + sessions. | Back End + Front End teams |
| Task 16 | 5/5/15 | 5/10/15 | 5 | Create a notification system to notify followers of new music + notify when you get a personal message. | Back End team |
| Task 17 | 5/10/15 | 5/12/15 | 2 | Usability testing, White box testing, Unit testing, Functional Testing | Back End + Front End teams |
| Task 18 | 5/12/15 | 18/5/2015 | 7 | Reserved for testing + Any changes that may need to be made as a result for testing | Back End + Front End teams |
| Task 19 | 18/5/2015 | 18/5/2015 | 1 | Submit project!!! | Team |