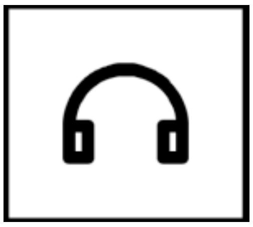
Dacian Spinu, Leonte Cristian, Condur Bogdan
Coordinator teacher - Buraga Sorin

MuST
Music Smart Assistant Technical Report

1. Objective

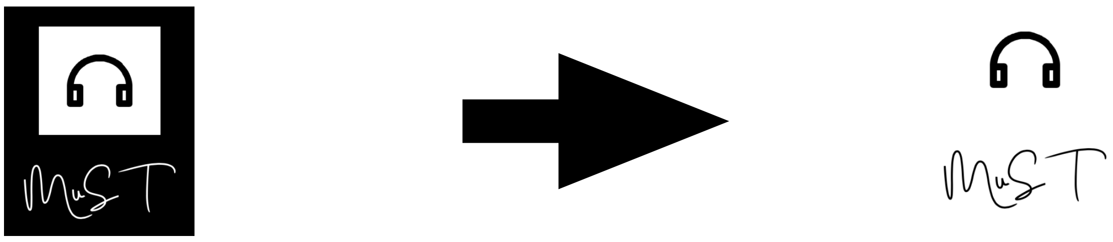
This report examines the design considerations that we took into account when creating the MuST web application, what challenges and problems we had to overcome from a user interaction and user experience point of view. Also, it brings in attention technical data about how the application is going to get the necessary data to be able to work and how that data is consumed by the application

2. Visual Identity and branding of MuST



MuST starts off as a service that acts on its own given a set of input data, collects that data, processes it and gives the user a set of personalised music playlists, based on the inputted data. This process we decided to incorporate into the box surrounding the set of headphones.

In the process of creating the logo we realized that we can achieve, using the negative space that the box provides, an inversed version of the logo. So, from the original black background with a white box we can create an inversed version with a white background but with black text. This will give us, in different use cases, the same image of the brand, but with the same meaning behind it.



3. Overall design considerations

Since MuST is going to be a web based application we had to create a design that can be implemented with existing web technologies and also to ensure a high level of compatibility between modern browser at development time.

Our main target in this process was a design that made sense being developed with HTML5 and CSS3 and little to none proprietary browser technologies from Google, Apple or other browser implementations. When we mention these proprietary browser technologies we mainly refer to CSS3 properties from the WebKit or Mozilla CSS3 extensions (*-webkit-*[proprietary_css_property], *-moz-*[proprietary_css_property]).

We wanted to achieve a simple, yet powerful web design that made sense in today's plethora of applications, without introducing a lot of interaction non-sense or elements that might disturb the focus of the user.

The interaction between the user and the interface is straight forward, the steps to create the account are minimal, the input from the user is also minimal, and the steps for the user to see the automatically generated playlists, interact with them, generate new ones are also minimal.

4. User on-boarding

Modern web applications heavily rely on a lot of user data and every business tries to get as much input as possible in order to provide an output that is as accurate as possible. The main issue in this era is getting the user data in a graceful way, without harassing the user for it.

The registration flow consists in only three steps. These steps will ensure that the application has all the data that is necessary to create reliable recommendations for the user and based on the data gathered in this registration flow we can also provide the links between genres and artists, artists and albums, artists and concerts and so on.

4.1 User registration

The first screen in the registration process is the account creation. The user will have to input an email address, a password and a confirmation of the chosen password. The decision of confirming the password a second time comes from the need of making sure that the user knows exactly what he inputted. The next step will be an account creation confirmation. The user receives an email that contains an URL that redirects him to the application, sends an HTTP request to the server with the data encoded in the URL and when the response is received from the server the user is sent to the setup screen.

4.2 <u>User account - preferences setup</u>

After the confirmation of the account the user arrives in the preferences setup screen. In order to create an interface that is not cluttered but in the same time offers a lot of options for the user we decided to put each element in a bubble that the user can click and, in consequence, select as his preference.

In this way we provide a dynamic experience that is playful, yet useful and easy to understand for the user. If the albums, genres or artists that appear in the bubbles are not exactly in accordance with the user preferences, then the user interface gives the possibility to the user to search for his particular musical interest, whatever they may be.

When searching, the option bubbles will change with the content that was inputted and in this way the user input can be much accurate and helpful in the end. When selecting an option, its border will change its colour according to the category of the selected item.

After the user selects his preferred albums, artists or genres and the selection counts more than 5 items he can proceed further by clicking the “DONE” button. This action will send an HTTP POST request to the server with the information selected. When the success response has arrived we can redirect the user to the next step.

4.3 <u>User account - streaming account setup</u>

After the user has managed to choose and save his preferences the application will redirect him to the next step, which is logging in to a music streaming service. In our mockups we provided the options for the user to login into one of multiple services, including Apple Music, GooglePlay Music, Spotify and LastFM.

At the time of writing this technical specification the only services that provide a developer API to integrate in web applications are Spotify and LastFM. Apple Music offers an SDK only for the iOS platform using the Swift Language and the music service that Google provides doesn’t even have an API.

However, LastFM and Spotify provide very complete web APIs to integrate in web applications and for the data that we need and the amount of users they already have we think that these services could complement the data that we already have at this point in a very nice way.

Both of these services require a developer account for each of these platforms in order to access the APIs. When registering for a developer account we will have access to all the private and the public keys. Both of these platform provide the same type of integration in the sense that when trying to login an user via an external application that consumes their web APIs will provide us with a login page and if the login in the platform is successful we will redirect the user into the next step.

Of course, we don't intend that all of our users should have an account on any of these platforms, so if they don't want to sign in they have the possibility of skipping this step.

5. Home page

After the user has logged in he is redirected to the home page, which, as the user is logged in, will be the main entry point in the web application.

In the home page we display a list of personalised playlists based on what the user has selected in the preferences setup page and the data that we get from the LastFM or Spotify web APIs (recently played songs, recently liked artists, smart playlists that are created by the before mentioned platforms, etc).

Each playlists has a short description of the reasoning that stood for its creation. For example, if we detect that the user set up in his preferences a lot of pop-rock bands we can create a pop-rock playlists that includes tracks from those artists.

Also, he has the possibility to create new playlists on demand just by clicking a button, which will randomise the genre, artist, album or other data that we can base our suggestion on, and will create a playlists for him.

6. Data provenance

The data for the preferences page setup comes from a free and online available musical database called MusicBrainz. According to the official webpage of the service, it is "an open music encyclopedia that collects music metadata and makes it available to the public."

This service, besides the fact that it is free and open-source, offers us an immensely valuable tool, which is a client library. They provide multiple versions for a multitude of programming languages like node.js or python.

Given the fact that most of the data that we will store about the playlists will be stored in a specific format, which is JSON Shareable Playlist Format, for us it will make sense to create a node.js backend, because the client library for node returns the responses in JSON format.

7. Architecture of the system

This chapter will touch the architecture of the system, how it will work internally and how it will use external data sources as Spotify or LastFM APIs and the open source database called MusicBrainz. Also we will explain the different modules of the backend application and also the frontend web application how they will communicate with each other through a REST API.

7.1 Web backend project technology details

The backend application will act as a RESTful API that will serve data to a frontend client via the HTTP protocol. All the calls are going to be authenticated but from this rule we will except the calls used for the authentication process.

The base language for the API will be NodeJS and all of its server functionality will be written using the ExpressJS framework. This framework will allow us to create in a very simplistic way a powerful and reliable API that can be easily opened to other sources other than our application with minimal effort. On top of ExpressJS we will use another very specific API framework called LoopBack. The main advantage of this framework is that it will instantiate a web api with all the user authentication processes in place from start and it has a lot of database connectors available to use. Also it provides a CLI (Command Line Interface) for creating the project but also for the creation of data models. Also, it setups automatically a Swagger API documentation that will be very helpful when developers create the frontend application and can read the documentation that contains all the proper documentation, with each model highlighted independently and all the API calls properly explained.

On the database we will use mainly MongoDB to store all the custom data in the application but we will also connect with the MusicBrainz database via the node api client.

7.1.2 Web backend project modules presentation

The web backend project being based on the LoopBack framework we will use the default project structure. At the root level we will find the `node_modules` directory which will include all our libraries, the `server` directory in which we will find all the configuration of the server application and the main entry point of the node app, which is the `server.js` file. After this we will find the `client` folder in which we can store the frontend client, but for that we will create another project that is standalone.

Because we don't want to store our models and their logic in the `common` folder that will make them available to the `client` folder and by extension to the client app that can be generated by the server we will store them in the server folder. In the sub-folder `models` we will register all our models which take the form of a JSON file like `model.json` and for each model we will have a JavaScript file that takes the form of `model.js` and will contain all the remote methods that we will write for that model.

All the data structures will be created with the help of LoopBack's CLI, like the lb model <model-name>

7.1.3 Users module

The users module will be created automatically by the generated LoopBack project and will contain all the functionality needed to build our application from an authentication point of view. Some of the functionality provided is the creation of the account, resetting the password and log in the account. Examples of the APIs provided :

```
1. POST /Users - Creates a new user
       User {
             realm (string, optional),
             username (string, optional),
             email (string),
             emailVerified (boolean, optional),
             id (number, optional)
       }
```

```
{
        “email”: “example@mail.com“,
        “password”: “0000”

}
```

will respond with a `200 OK` and details after the newly created account

```
2. POST /Users/login - Logins the user
        credentials : {
                email: “example@mail.com",
                “password”: “0000”
        }
```

will respond with a `200 OK` and the access token that we will now use to sign all the next requests.

7.1.4. Genres module

The `genres` module data structure is going to be very simple, will contain an `id` and its `name`. We will this model mainly in the `playlists` module, `albums` module and for the account setup screen.

In the MongoDB database the model will look like this :

```
{
    "name": "string",
    "id": "string"
}
```

When the user gets in the preferences setup screen we will fetch this data via the API with a GET request in the next form :

```
GET /genres - 200 OK
[
  {
    "name": "string",
    "id": "string"
  }
]
```

7.1.5 Artists module

The `artists` module data structure will be saved in the database as the following structure and we will use it in the `playlists` module and also to show them to the user in order to save them as preferred artists.

```
{
    "name": "string",
    "type": "string",
    "members": [
      {
           “name” : “string”,
           “enter_date” : “date”,
           “active” : “boolean”
      }
    ],
    "id": "string"
  }
```

The type key can have values of “`band`”, “`singer`”. The `members` key will store current and ex members of the artist if its type is “`band`”. Each member has a `name`, a date at which he joined the band (`enter_date`) and if he is a still an active member of the band (`active`).

When retrieving the artists lists for the preferences setup screen we will fetch the data with a GET requests to the artists endpoint as follows.

```
GET /artists - 200 OK
[
  {
    "name": "string",
    "type": "string",
    "members": [
      {
           “name” : “string”,
           “enter_date” : “date”,
           “active” : “boolean”
      }
    ],
    "id": "string"
  }
]
```

7.1.6 Albums module

The `Albums` module data structure will be saved in the database in the following structure and it will be also used in the preferences account setup.

```
{
    "name": "string",
    "release_date": "date",
    "artists": [
      {Artist Object}
    ],
    "record_label_name": "string",
    "id": “string”,
    "gender": "string"
  }
```

As in the other modules, in order to get all of the items in the preferences setup screen we will perform a GET request as it follows

`GET /albums` - that will respond with a `200 OK` status if no other server-side error occurs and will return an array of `album` objects.

7.1.7. Preferences module and automatically intelligent playlist generation

The preferences module is going to be crucial in the whole application environment due to its importance in the automatic playlist generation.

This functionality will need as much data as possible in order to create accurate recommendations.

It will take the data that we store in the database via constantly calling the node API that MusicBrainz has and will populate the genres, artists and albums tables. After this we prompt these to the user when creating or updating the preferences.

When the user sends these back to the API in order to save them we have a basis for searching tracks based on a combination of liked albums, genres or artists.

Then, if the user logs into Spotify we can use parts of the data that this service provides us to

1. Prompt the user with Spotify Generated playlists (`GET /v1/me/playlists`)
2. Gather more data about user's preferred albums, genres or artists (`GET /v1/me`)
3. Improve experience of the user by creating new playlists using related information from Spotify about already preferred artists (`v1/artists/{id}/related-artists`)

The preferences data will be stored in the MongoDB database in a structure like the following, and when the current user logs in we will be able to track his preferences based on the userId that generated that playlists.

```
{
  "generated_by": "string",
  "albums": [
    "string"
  ],
  "artists": [
    "string"
  ],
  "genre": [
    "string"
  ],
  "id": "string"
}
```

The albums, artists and genre keys will only contain id references to the original items in the database and the generated_by key represents the user who has this particular preference setup.

7.1.9 Tracks

This module comes as a necessity to keep the track entity and playlist entity separated logically but to be able to integrate the track information into the playlist.

The data structure of this module follows the specifications provided by the XSPF community in the form of the JSON XPF (or JSPF) specifications.

The data looks as it follows

```
{
    "name": "string",
    "release_date": "date",
    "album": "string",
    "duration": 0,
    "artist": {Artist Object},
    "id": "string"
  }
```

7.1.8 Playlists module

The playlists module will be the main component with which a user will interact. The user will be able to generate playlists based on the configured preferences, delete them, see their details.

As we've said before, the playlist data will follow the JSPF specification and its data structure will look as follows

```
{
    "title": "string",
    "creator": "string",
    "tracks": [{Track Object}],
    "id": “string”,
    “owner”: “string”
  }
```

The creator key will hold the entity that created the playlist, if it is the user or a third party, like Spotify. The title will be generated based on the criteria that the application will select to create it upon. The `owner` key represents the user id that generated that playlists.

When the user logs back in the application we will fetch data from Spotify to gather new playlists that the service might've created based upon its own criteria and we will prompt those as well in the playlists view.

8. Frontend application specifications

The web-client that we has to be implemented will be addressed in summary here and it will only touch a few key points, like technologies that need to be used, how it will consume the API that we provide and other technical details like architecture (or folder structure).

With the help of frontend application we will also touch a few points in the data flow of the overall solution.

8.1 Technologies and architecture

The frontend scene has suffered a lot of change in the past few years, with a lot of emerging frameworks rising and package management solutions that are vastly superior than anything this eco-system was dreaming of.

But in the last few years a particular technology made its way into the market and it seems to have a lot of benefits that we can use. This technology is called React and is provided by Facebook, it is well documented and also very fast and provides extension capabilities for the solution into the mobile applications territory via the framework that is called React-Native.

We will establish a feature-oriented architecture because in this way we can contain the necessary components for a feature enclosed and the ones that are more common and are used in multiple screens or bigger components will be placed closer to the root of the project.

So, the first concept is that of the Scene. A scene is a view of the application and it can contain sub-scenes or other components and also services to communicate with the API provided. The components that are contained in scenes that are "brothers" with other scenes cannot be used in any other scene other than the one where it was defined. If we need to do this we will extract that component higher so that it is available to both scenes.

A component is a piece of view or logic that handles only one thing. For example, a button component will handle only click events, it will not trigger per-se the call to the server for a specific functionality, rather it will dispatch an event or it will call inside the logic component that handles that part.

Services are logic components, like a data service that only handles communication with the server. For example, in the login scene we can have a LoginService that will handle the login calls and will dispatch events to other logic components, like a generic application router, that will respond to that event and will change the scene to the preferences setup scene.

8.2 Data flow

We will try to create a generic step-by-step flow of how the data is used throughout the frontend and backend applications.

Pre-requisite data flow

1. Server Application instantiates all the data models
2. Server Application constantly polls the MusicBrainz API to fetch and populates the MongoDB database. We want to do this in order to have a copy on our side in case the MusicBrainz service fails or becomes inactive.

Generic Flow

1. Client application sends a request to create a new user
2. Server application creates a new user in the MongoDB database and sends an email that contains a confirmation url
3. Client accesses the confirmation url and that will trigger a POST request to the server application to confirm the account. Now the user is able to log in
4. Clients sends an object containing “user” and “password”
5. Server checks if the combination is correct and sends back a Bearer Token that the frontend application will now use in order to sign the requests.

6. Client can begin setup of the screens and generates three get requests, one to fetch the albums, one to fetch the artists and one to fetch the genres.

GET /artists - responds with an array of artists

GET /albums - responds with an array of albums

GET /genres - responds with an array of genres

7. Server sends the responses with the data requested and the frontend application can now setup the preferences

8. Frontend application creates a POST request with all the artists, albums and genres to save the new preference.

```
POST /preferences
{
  "generated_by": "string",
  "albums": [
    "string"
  ],
  "artists": [
    "string"
  ],
  "genre": [
    "string"
  ],
  "id": "string"
}
```

9. The server application receives the data in the preference object, saves it in the database and proceeds to create 5 playlists based on the preferences. In order to do this it will fetch the tracks table for tracks that contain that information and will create the playlist and save them into the database. After this the server will respond with 200 OK.

10. When server has responded the client can check all the playlists in the playlists screen.

11. If the user wants to create another playlist they will hit the “create a new playlist” button and that will generate a new process of creating a playlist that uses the same algorithm used by the 9th step.

12. When the user wants to remove a part of his preferences, lets say an album, they will access the preferences page and will hit the “X” icon and that will trigger a DELETE request for that resource.

13. When the DELETE is handled first the server application will check all the playlists generated by the user that contain that resource and will remove the tracks from the playlists and then will remove the resource from the user’s preferences. After this it will send a 200 OK response.