

CS 157A: AMDb (AudioMedia Database)

Names: Jonathan Manzano, Averil Tanlimco, Paul Valdez, and Katrina Weers

IDs: 007938580, 015612922, 008599578, 015537730

Emails: jonathan.manzano@sjsu.edu, averil.tanlimco@sjsu.edu,
paul.valdez@sjsu.edu, katrina.weers@sjsu.edu

San Jose State University

CS 157A Section 01

May 10, 2024

Goals and Application Description

Problem Statement:

Currently, users are unable to access audio media information in a centralized place. Users have to navigate through multiple websites in order to access details about their favorite songs, podcasts, and audiobooks. Oftentimes they find audio media from streaming apps like Spotify and iTunes, but need to research their details on Wikipedia or Songfacts. Our solution is a user-friendly web application, similar to IMDb, that allows users to easily access information about various forms of audio media.

Goals:

Our goal is to create a web application that allows users to retrieve audio media information from our database. The user will be able to choose what type of audio media they want to retrieve (Song, Music Artist, Record Label, Music Release, Podcast Host, Podcast, Podcast Episode, Audio Book, Author, Narrator, and Publisher) and must add specifications to narrow the results they receive. This will query our database (AMDb) and display the matching results to the user.

Application Description:

The database stores 11 types of Audio Media / Audio Media related entities. The user will be able to retrieve each entity's single-valued and multi-valued attributes from the database.

Entities:

- **Song:** A Song has an ID, title, genre, tempo, length (in minutes), and explicit status. Each Song must belong to only one Music_Release. Each Song may be recorded by any number of Artists.
- **Music_Artist:** A Music_Artist has an ID and a name. Each Music_Artist can be signed to any number of Record_Labels. Each Music_Artist can record any number of songs. Each Music_Artist can have any number of Music_Releases.
- **Record_Label:** A Record_Label has an ID and name. Each Record_Label can release any number of Music_Releases. Each Record_Label can have any number of Music_Arists signed to it.
- **Music_Release:** A Music_Release has an ID and release_title. Each Music_Release can be released under any number of Record_Labels. Each Music_Release can have any number of Songs.
- **Podcast_Host:** A Podcast_Host has an ID and name. Each Podcast_Host can host any number of Podcasts.

- **Podcast:** A Podcast has an ID, title, url, description, and explicit status. Each Podcast can have any number of Hosts. Each Podcast can have any number of Podcast_Episodes.
- **Podcast_Episode:** A Podcast_Episode has an ID and title. Each Podcast_Episode must belong to only one Podcast.
- **Audio_Book:** An Audio_Book has an ID, title, length, and release date. Each Audio_Book can be published by any number of Publishers. Each Audio_Book can be written by any number of Authors. Each Audio_Book can be narrated by any number of Narrators.
- **Author:** An Author has an ID and name. Each Author can write any number of Audio_Books.
- **Narrator:** A Narrator has an ID and name. Each Narrator can narrate any number of Audio_Books.
- **Publisher:** A Publisher has an ID and name. Each Publisher can publish any number of Audio_Books.

Application/Functional Requirements and Architecture

Application/Functional Requirements:

The user selects an option from the dropdown menu (Song, Music Artist, Record Label, Music Release, Podcast Host, Podcast, Podcast Episode, Audio Book, Author, Narrator, and Publisher) and is presented with textboxes and checkboxes to specify the characteristics of the results they want to receive. After inputting their specifications and clicking the submit button, the results matching the specifications are displayed underneath. If no specifications are made, then no results are returned. Each result contains information about the requested type (ie. Its single-valued and multi-valued attributes).

Functional Requirements For Each Entity:

- **Song:** The user can input strings for Song Title and Genre. The user can specify a range of numbers for the Song's Tempo and Length. The user can select a checkbox for showing only explicit results. If the checkbox is not clicked, it will show both explicit and non-explicit results. Each of the string inputs (Song Title, Genre) would be used as a partial match and are not case sensitive. For example, if the user submits "pop" as the genre, all Songs that contain the word "pop" anywhere in its Genre attribute would be returned ("pop", "cantopop", "k-pop", etc.). Input validation is included for the Song's Tempo and Length. If the user enters a minimum value that is larger than its maximum value, the website will notify the user and set the maximum number to the larger minimum number. Each returned Song result would display its Song_Id, Song_Title,

Genre, Tempo, Length (in Xminutes Yseconds Zmilliseconds format), Explicit status, the title of the Music_Release it belongs to, and the names of the Music_Artists that recorded it.

- **Music_Artist:** The user can input a string for Artist Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “tentacion” as the name, then all Music_Artists whose name contains “tentacion” anywhere in its Artist_Name attribute would be returned (“XXXTentacion”, “Tentación”, etc.). Each returned Music Artist result would display its Artist_ID, Artist_Name, the names of the Record_Labels the artist is signed to, the titles of the Music_Releases the artist created, and the titles of the Songs the artist recorded.
- **Record_Label:** The user can input a string for Record Label Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “records” as the name, then all Record Labels whose name contains “records” anywhere in its Label_Name attribute would be returned (“Capitol Records”, “Atlantic Records”, etc.). Each returned Record Label result would display its Label_ID, Record Label Name, the names of the Music Releases released by the label, and the names of the Music_Artists signed to the label.
- **Music_Release:** The user can input a string for Release Title. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “foggy” as the title, then all Music Releases whose title contains “foggy” anywhere in its Release_Title attribute would be returned (“foggy afternoons”, “Foggy Mountain Banjo”, etc.). Each returned Release Title result would display its Release_ID, Release_Title, the names of the Record_Labels that released it, and the names of the Music_Artists that created it.
- **Podcast_Host:** The user can input a string for Host Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “bread” as the name, then all Podcast_Hosts whose name contains “bread” anywhere in its Host_Name attribute would be returned (“Glennette Goodbread”, “Dean Whitbread”, etc.). Each returned Podcast_Host result would display its Host_ID, Host Name, and the names of the Podcasts the host runs.
- **Podcast:** The user can input a string for Podcast Title and select a checkbox to show only explicit results. If the checkbox is not selected, then the results will include both explicit and non-explicit results. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “ready” as the title, then all Podcasts whose title contains “ready” anywhere in its Podcast_Title attribute would be returned (“Painkiller Already”, “Ready, Set, Knit”, etc.). Each returned Podcast result would display its Podcast_ID, Podcast_Title, Url, Explicit status, Description, and names of the

Podcast_Hosts.

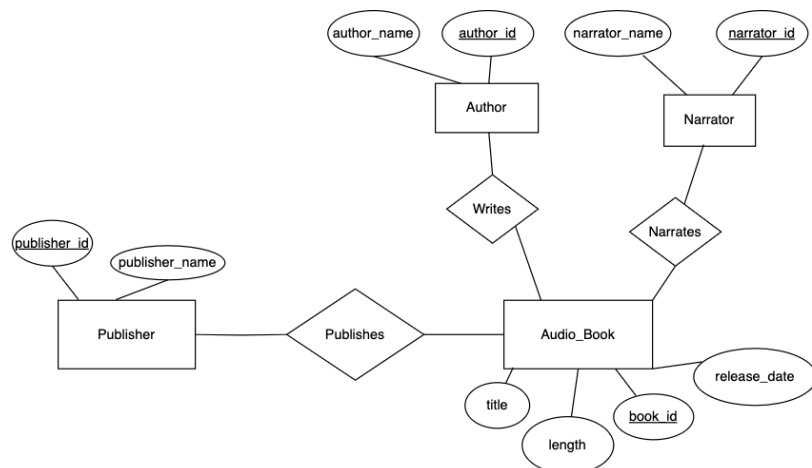
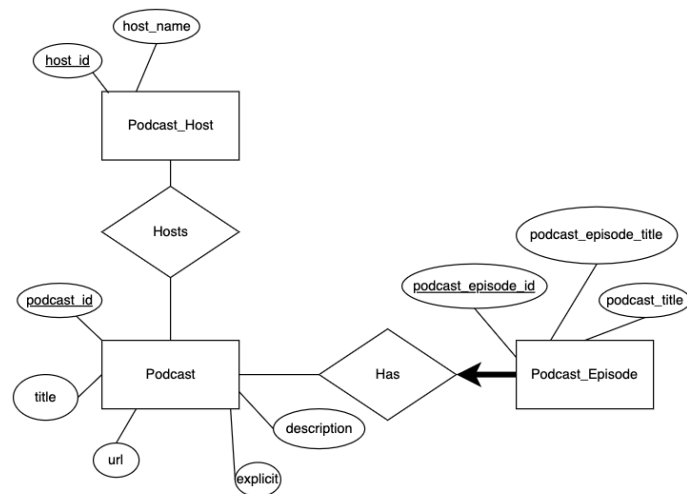
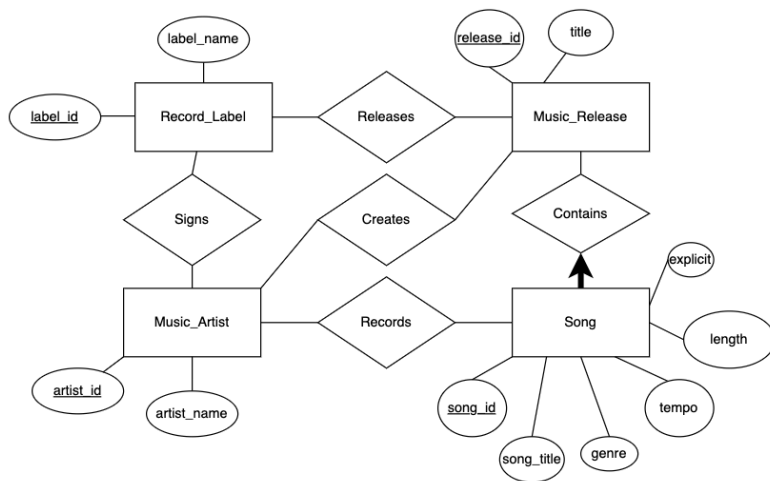
- **Podcast_Episode:** The user can input a string for Podcast Episode Title. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “Episode 10 - You” as the title, then all Podcast_Episodes whose name contains “Episode 10 - You” anywhere in its Podcast_Episode_Title attribute would be returned (“Episode 10 - You Think You’re Better Than Me?”, “Episode 10 - You’re Only A Human”, etc.). Each returned Podcast_Episode result would display its Podcast_Episode_ID, Podcast_Episode_Title, and title of the Podcast it belongs to.
- **Audio_Book:** The user can input a string for Audiobook Title. The user can specify a range of numbers for the Audio_Book’s Length. The user can enter in or select a minimum Release Date. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “percy jackson” as the title, then all Audio_Books that contain the word “percy jackson” anywhere in its Book_Title attribute would be returned (“The Lightning Thief: Percy Jackson, Book 1”, “Percy Jackson and the Battle of the Labyrinth”, etc.). Input validation is included for the Audio_Book’s Length. If the user enters a minimum value that is larger than its maximum value, the website will notify the user and set the maximum number to the larger minimum number. Input validation is also included for the minimum Release Date. The date is chosen from an HTML date input type ensuring that no erroneous dates can be entered. The user inputted release date is used as the minimum date, meaning that all the results will be equal to or later than the given release date. For example, if the user submits “01/01/2020” as the release date, then all results will be from “January 1, 2020” or later. Each returned Audio_Book result would display its Book_ID, Book_Title, Length (in minutes), Release_Date, names of the book’s Publishers, names of the book’s Authors, and names of the book’s Narrators.
- **Author:** The user can input a string for Author Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “Riordan” as the name, then all Authors whose name contains “Riordan” anywhere in its Author_Name attribute would be returned (“RickRiordan”, “MichaelRiordan”, etc.). Each returned Author result would display its Author_ID, Author_Name, and titles of the Audio_Books they’ve written.
- **Narrator:** The user can input a string for Narrator Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “Bernstein” as the name, then all Narrators whose name contains “Bernstein” anywhere in its Narrator_Name attribute would be returned (“JesseBernstein”, “DanielleBernstein”, etc.). Each returned Narrator result would display its Narrator_ID, Narrator_Name, and titles of the Audio_Books they’ve narrated.

- Publisher:** The user can input a string for Publisher Name. The string input would be used as a partial match and is not case sensitive. For example, if the user submits “ha” as the name, then all Publishers whose name contains “ha” anywhere in its Publisher_Name attribute would be returned (“HarperCollins”, “Hachette Livre”, etc.). Each returned Publisher result would display its Publisher_ID, Publisher_Name, and titles of the Audio_Books they’ve published.

Architecture:

We used a 3-tier architecture. The user interacts with the HTML User Interface (Tier 1). The user input gets sent to the Java Application, where it gets interpreted and converted into SQL statements (Tier 2). The SQL statements are sent to the MySQL database (Tier 3) and ResultSets are returned to the Java Application (Tier 2). The Java Application stores the attributes of each result in a LinkedHashMap, and stores all the LinkedHashMaps (ie. all the processed results) in an ArrayList. The HTML file then renders the results that were stored in the ArrayList for the user to see (Tier 1).

ER Data Model



Database Design

Tables (Relational Schema) and Normalization to BCNF:

Relations

- Record_Label(label_id: Integer, label_name: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** $FD1 = (Label_Id \rightarrow Label_Name)$
 - $FD1$ is a functional dependency because $Label_Id$ is unique. A unique id only shows up once in its column. Therefore, every instance of the id (which is only one instance) will result in the same values for the non candidate key attributes.
 - **Candidate Key:** $Label_Id$
 - $Label_Id$ is not functionally dependent on any other attribute, meaning it must be in the candidate key. This means that $Label_Id$ and $Label_Name$ together cannot be a candidate key, since it would not be minimal. Thus, $Label_Id$ is the only candidate key for the $Record_Label$ relation.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute ($Label_Name$) is fully functionally dependent on the candidate key ($Label_Id$), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute ($Label_Name$) is transitively dependent on a candidate key ($Label_Id$), as seen in the listed functional dependencies. $Label_Name$ is fully and directly dependent on $Label_Id$.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, $(Label_Id \rightarrow Label_Name)$. The LHS contains a candidate key ($Label_Id$). Thus, this relation is in BCNF.
- Music_Artist(artist_id: Integer, artist_name: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** $FD1 = (Artist_Id \rightarrow Artist_Name)$
 - $FD1$ is a functional dependency because $Artist_Id$ is unique.
 - **Candidate Key:** $Artist_Id$
 - $Artist_Id$ is not functionally dependent on any other attribute, meaning it must be *in* the candidate key. $Artist_Id$ functionally determines $Artist_Name$, which is the only other attribute in the relation, meaning $Artist_Id$ is a candidate key. $Artist_Id$ and $Artist_Name$ together also identify each value in the relation, but it is not minimal, since it contains a candidate key ($Artist_Id$). Thus, $Artist_Id$ is the only candidate key for the $Music_Artist$ relation.

- **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute (Artist_Name) is fully functionally dependent on the candidate key (Artist_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute (Artist_Name) is transitively dependent on a candidate key (Artist_Id), as seen in the listed functional dependencies. Artist_Name is fully and directly dependent on Artist_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, (Artist_Id -> Artist_Name). The LHS contains a candidate key (Artist_Id). Thus, this relation is in BCNF.
- Music_Release(release_id: Integer, release_title: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Release_Id -> Release_Title)
 - FD1 is a functional dependency because Release_Id is unique.
 - **Candidate Key:** Release_Id
 - Release_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute (Release_Title) is fully functionally dependent on the candidate key (Release_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute (Release_Title) is transitively dependent on a candidate key (Release_Id), as seen in the listed functional dependencies. Release_Title is fully and directly dependent on Release_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, (Release_Id -> Release_Title). The LHS contains a candidate key (Release_Id). Thus, this relation is in BCNF.
 - Song(song_id: Integer, song_title: String, genre: String, tempo: Integer, length: Integer, explicit: Boolean, music_release_id: Integer (FK))
 - *Foreign Key:*
 - Music_Release_Id is a foreign key to Release_Id in Music_Release
 - *BCNF Normalization:*

- **Functional Dependencies:** FD1 = (Song_Id -> Song_Title), FD2 = (Song_Id -> Genre), FD3 = (Song_Id -> Tempo), FD4 = (Song_Id -> Length), FD5 = (Song_Id -> Explicit), FD6 = (Song_Id -> Music_Release_Id)
 - FD1 - FD6 are functional dependencies because Song_Id is unique.
 - Song_Title, Genre, Tempo, Length, Explicit, and Music_Release_Id do not determine any of the other attributes. That is, for any given value of these attributes, the rest of the attributes may have any value. Therefore, they are not determinants.
- **Candidate Key:** Song_Id
 - Song_Id is the candidate key because it is not functionally dependent on any other attribute (so it must be in the candidate key) and it leads to all other attributes in the relation.
- **Check 1NF:** Since this is a relational model, it is already in 1NF.
- **Check 2NF:** Every non-candidate key attribute (Song_Title, Genre, Tempo, Length, Explicit, Music_Release_Id) is fully functionally dependent on the candidate key (Song_Id), as seen in the listed functional dependencies.
- **Check 3NF:** No non-candidate key attribute is transitively dependent on a candidate key (Song_Id), as seen in the listed functional dependencies. All non-candidate key attributes are fully and directly dependent on Song_Id.
- **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 6 functional dependencies. The LHS for each of these dependencies contains a candidate key (Song_Id) as seen in the list of functional dependencies. Thus, this relation is in BCNF.
- Signs(label_id: Integer (FK), artist_id: Integer (FK))
 - *Foreign Keys:*
 - Label_Id is a foreign key to Label_Id in Record_Label
 - Artist_Id is a foreign key to Artist_Id in Music_Artist
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Records(artist_id: Integer (FK), song_id: Integer (FK))
 - *Foreign Keys:*
 - Artist_Id is a foreign key to Artist_Id in Music_Artist
 - Song_Id is a foreign key to Song_Id in Song
 - *BCNF Normalization:*

- **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
- **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Releases(label_id: Integer (FK), release_id: Integer (FK))
 - *Foreign Keys:*
 - Label_Id is a foreign key to Label_Id in Record_Label
 - Release_Id is a foreign key to Release_Id in Music_Release
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Creates(artist_id: Integer (FK), release_id: Integer (FK))
 - *Foreign Keys:*
 - Artist_Id is a foreign key to Artist_Id in Music_Artist
 - Release_Id is a foreign key to Release_Id in Music_Release
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Contains(release_id: Integer (FK), song_id: Integer (FK))
 - *Foreign Keys:*
 - Release_Id is a foreign key to Release_Id in Music_Release
 - Song_Id is a foreign key to Song_Id in Song
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Podcast_Host(host_id: Integer, host_name: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Host_Id -> Host_Name)
 - FD1 is a functional dependency because Host_Id is unique.
 - **Candidate Key:** Host_Id

- Host_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal.
 - Check 1NF: Since this is a relational model, it is already in 1NF.
 - Check 2NF: Every non-candidate key attribute (Host_Name) is fully functionally dependent on the candidate key (Host_Id), as seen in the listed functional dependencies.
 - Check 3NF: No non-candidate key attribute (Host_Name) is transitively dependent on a candidate key (Host_Id), as seen in the listed functional dependencies. Host_Name is fully and directly dependent on Host_Id.
 - Check BCNF: Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, (Host_Id \rightarrow Host_Name). The LHS contains a candidate key (Host_Id). Thus, this relation is in BCNF.
- Podcast(podcast_id : Integer, podcast_title: String, url : String, explicit : Boolean, description : String)
 - *BCNF Normalization:*
 - Functional Dependencies: FD1 = (Podcast_Id \rightarrow Podcast_Title), FD2 = (Podcast_Id \rightarrow Url), FD3 = (Podcast_Id \rightarrow Explicit), FD4 = (Podcast_Id \rightarrow Description)
 - FD1 - FD4 are functional dependencies because Podcast_Id is unique.
 - Podcast_Title, Url, Explicit, and Description cannot guarantee the same values for any of the other attributes. Two podcasts may have the same name, but different urls, explicit status, description, and id. Two podcasts may be located on the same page, making them have the same url, but different titles, explicit status, description, and id. Explicit status and description are also clearly not determinants for any of the other attributes.
 - Candidate Key: Podcast_Id
 - Podcast_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal. There are no other attributes that can lead to all the other attributes, so Podcast_Id is the only candidate key.
 - Check 1NF: Since this is a relational model, it is already in 1NF.
 - Check 2NF: Every non-candidate key attribute (Podcast_Title, Url, Explicit, Description) is fully functionally dependent on the candidate key (Podcast_Id), as seen in the listed functional dependencies.
 - Check 3NF: No non-candidate key attribute is transitively dependent on a candidate key (Podcast_Id), as seen in the listed functional dependencies. All non-candidate keys are fully and directly dependent on Podcast_Id.

- **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 4 functional dependencies. The LHS of each FD contains a candidate key (Podcast_Id) as seen in the listed functional dependencies. Thus, this relation is in BCNF.
- Podcast_Episode(podcast_episode_id : Integer, podcast_episode_title : String, podcast_id: Integer (FK))
 - *Foreign Key:*
 - Podcast_Id is a foreign key to Podcast_Id in Podcast
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Podcast_Episode_Id → Podcast_Episode_title), FD2 = (Podcast_Episode_Id → Podcast_Id)
 - FD1 and FD2 are functional dependencies because Podcast_Episode_Id is unique.
 - For every instance of a value in Podcast_Episode_Id (each value has only one instance), the Podcast_Episode_Title and Podcast_Id will be the same. But the same cannot be said the other way around, as the same Podcast_Id can have different Podcast_Episode_Id and Podcast_Episode_Title values associated with it. Similarly, two podcast episodes may be titled the same name while belonging to different Podcasts, making it not a determinant of either the Podcast_Episode_Id nor the Podcast_Id.
 - **Candidate Key:** Podcast_Episode_Id
 - Podcast_Episode_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal. Also, no other attribute can lead to all the other attributes in the relation, making Podcast_Episode_Id the only candidate key.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute (Podcast_Episode_Title, Podcast_Id) is fully functionally dependent on the candidate key (Podcast_Episode_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute (Podcast_Episode_Title, Podcast_Id) is transitively dependent on a candidate key (Podcast_Episode_Id), as seen in the listed functional dependencies. Both non-candidate keys are fully and directly dependent on Podcast_Episode_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.

- We have 2 functional dependencies. The LHS of both FDs contain a candidate key (Podcast_Episode_Id) as seen in the listed functional dependencies. Thus, this relation is in BCNF.
- Hosts(host_id : Integer (FK), podcast_id : Integer (FK))
 - *Foreign Keys:*
 - Host_Id is a foreign key to Host_Id in Podcast_Host
 - Podcast_Id is a foreign key to Podcast_Id in Podcast
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Publisher(publisher_id: Integer, publisher_name: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Publisher_Id -> Publisher_Name)
 - FD1 is a functional dependency because Publisher_Id is unique.
 - **Candidate Key:** Publisher_Id
 - Publisher_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute (Publisher_Name) is fully functionally dependent on the candidate key (Publisher_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute (Publisher_Name) is transitively dependent on a candidate key (Publisher_Id), as seen in the listed functional dependencies. Publisher_Name is fully and directly dependent on Publisher_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, (Publisher_Id -> Publisher_Name). The LHS contains a candidate key (Publisher_Id). Thus, this relation is in BCNF.
- Audio_Book(book_id:Integer, book_title: String, length: Integer, release_date: Date)
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Book_Id -> Book_Title), FD2 = (Book_Id -> Length), FD3 = (Book_Id -> Release_Date)
 - FD1 - FD3 are functional dependencies because Book_Id is unique.

- Multiple books may have the same title but different lengths and release_dates, meaning it can't be a determinant. Multiple books can be released on the same day, which means it also isn't a determinant. Multiple different books can have the same length, which means it isn't a determinant.
 - **Candidate Key:** Book_Id
 - Book_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal. Also, no other attribute can lead to all the other attributes in the relation, making Book_Id the only candidate key.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute is fully functionally dependent on the candidate key (Book_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute is transitively dependent on a candidate key (Book_Id), as seen in the listed functional dependencies. All non-candidate keys are fully and directly dependent on Book_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 3 functional dependencies. The LHS of the three FDs contain a candidate key (Book_Id) as seen in the listed functional dependencies. Thus, this relation is in BCNF.
- **Author(author_id:Integer, author_name: String)**
 - *BCNF Normalization:*
 - **Functional Dependencies:** FD1 = (Author_Id -> Author_Name)
 - FD1 is a functional dependency because Author_Id is unique.
 - **Candidate Key:** Author_Id
 - Author_Id is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute (Author_Name) is fully functionally dependent on the candidate key (Author_Id), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute (Author_Name) is transitively dependent on a candidate key (Author_Id), as seen in the listed functional dependencies. Author_Name is fully and directly dependent on Author_Id.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, (Author_Id -> Author_Name). The LHS contains a candidate key (Author_Id). Thus, this relation is in BCNF.

- Narrator(narrator_id:Integer, narrator_name: String)
 - *BCNF Normalization:*
 - **Functional Dependencies:** $FD1 = (Narrator_Id \rightarrow Narrator_Name)$
 - $FD1$ is a functional dependency because $Narrator_Id$ is unique.
 - **Candidate Key:** $Narrator_Id$
 - $Narrator_Id$ is the candidate key because it is not functionally dependent on any other attribute (meaning it must be in the candidate key) and it is minimal.
 - **Check 1NF:** Since this is a relational model, it is already in 1NF.
 - **Check 2NF:** Every non-candidate key attribute ($Narrator_Name$) is fully functionally dependent on the candidate key ($Narrator_Id$), as seen in the listed functional dependencies.
 - **Check 3NF:** No non-candidate key attribute ($Narrator_Name$) is transitively dependent on a candidate key ($Narrator_Id$), as seen in the listed functional dependencies. $Narrator_Name$ is fully and directly dependent on $Narrator_Id$.
 - **Check BCNF:** Every functional dependency's determinant contains a candidate key.
 - We have 1 functional dependency, $(Narrator_Id \rightarrow Narrator_Name)$. The LHS contains a candidate key ($Narrator_Id$). Thus, this relation is in BCNF.
- Publishes(publisher_id: Integer (FK),book_id:Integer (FK))
 - *Foreign Keys:*
 - $Publisher_Id$ is a foreign key to $Publisher_Id$ in *Publisher*
 - $Book_Id$ is a foreign key to $Book_Id$ in *Audio_Book*
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.
- Writes(author_id:Integer (FK),book_id:Integer (FK))
 - *Foreign Keys:*
 - $Author_Id$ is a foreign key to $Author_Id$ in *Author*
 - $Book_Id$ is a foreign key to $Book_Id$ in *Audio_Book*
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.

- Narrates(narrator_id:Integer (FK),book_id:Integer (FK))
 - *Foreign Keys:*
 - Narrator_Id is a foreign key to Narrator_Id in Narrator
 - Book_Id is a foreign key to Book_Id in Audio_Book
 - *BCNF Normalization:*
 - **Functional Dependencies:** The table only has two attributes and both are part of the primary key, meaning there are no functional dependencies.
 - **Check BCNF:** Since there are no functional dependencies, the relation is automatically in BCNF.

Major Design Decisions

Architecture:

We decided to use a 3-tier architecture. We chose to use a 3-tier architecture because it gives us more control over how the user can interact with the database. This way, the user is not making direct requests to the database; user input must get processed by the Java Application middleware, which has PreparedStatements that limit the types of queries that can be made.

Stack:

We chose to use HTML, Thymeleaf, and Javascript to develop the Client / User Interface (Tier 1), Java Spring Boot to develop the Application-Server / Middleware (Tier 2), and MySQL to develop the Data-Server (Tier 3).

We chose to use Thymeleaf because it works well with Java Spring Boot's MVC (Model, View, Controller) pattern. Spring Boot has a Model, which is used to store information that will be moved between the HTML User Interface and the Java Backend. We created the AMDBController, which contains the Get and Post methods for taking in user input and returning the updated HTML page (which is the View).

The reason why we used Java Spring Boot is because it automatically sets up the web server, allowing us to just focus on creating the Java classes (TranslatorService, AMDBController, DataConnection). Within our Java Application, we used java.sql's DriverManager and the JDBC API to make the connection to the MySQL database, create and execute PreparedStatements to make calls to the database, and process the ResultSets that were returned.

Indexing:

We decided to create an index for each entity table based on each table's primary key. The primary key is a good attribute to use for indexing because each value is unique. We also decided to create two indexes for the relationship tables, one for each column. The reason why we created these indexes is because most entities have multi-valued attributes, whose relationship is stored in a separate table. For example, a Song can be recorded by multiple Music_Artists; this multivalued attribute is represented as the Records table, which has a column

for Song_Id and Artist_Id. We make frequent queries to retrieve the multivalued attributes, which require joins between tables based on id and exact comparisons between ids. Therefore, given there are thousands of entries for most of the tables and we use the ids often, it made sense to create indexes to speed up query execution and to use the id columns to do so.

Implementation Details

Java Spring Boot Application:

Many of the files within the AMDBProject folder were automatically created by Java Spring Boot Initializer. The application.properties file is an automatically generated file that we updated to let Spring Boot know to use our MySQL database. The pom.xml file is an automatically generated file that contains the dependencies selected during the initial download of the Java Spring Boot Project. The dependencies used in this application are: Spring Web, MySQL Driver, JDBC API, Thymeleaf, and Lombok. The AMDBProjectApplication java file is an automatically generated class created by Spring Boot containing the main method that runs the application. The following files were fully written by the team: DatabaseConnection.java, TranslatorService.java, AMDBController.java, the java classes in ModelAttributes, home.html, create-AMDB.sql, drop-AMDB, and insert-AMDB.sql.

Java Packages:

- ***ModelAttributes:*** This package contains the classes used to take in user input. Each class contains attributes that store the user input. Each of these classes also use the Lombok package to create the getter methods for each attribute. These classes are: Song.java, RecordLabel.java, MusicArtist.java, MusicRelease.java, PodcastHost.java, Podcast.java, PodcastEpisode.java, AudioBook.java, Publisher.java, Author.java, and Narrator.java.
- ***AMDBProject:***
 - **DatabaseConnection:** This java file contains a static method called getDBconnection to establish a connection to the database. First, an instance of the Connection class named connection is initialized to null. The variable connection's value will update when a valid database is found. The method will try to find an instance of the database, specifically the JDBC, by using the class "Class" method forName method and passing 'com.mysql.cj.jdbc.Driver'. If the instance of the database does not exist, then it will catch the exception and returns the message: "JDBC Driver not found: " plus the caught exception. After the Class method forName returns the instance of the JDBC, three string variables are declared: dbUrl, dbUser, and dbPassword. dbUrl is initialized to the address to the database provided by the JBDC, jdbc:mysql://localhost:3306/AMDB. Variable dbUser and dbPassword are initialized the credentials to the database, "user" and "password" respectively. The variables dbUrl, dbUser, and dbPassword are

checked if they are properly assigned the right url and credentials, if not then an exception is thrown, caught and returns the error message, “Database credentials are not set in the environment variables.” The instance of the Connection class named connection is updated and assigned a Connection object that is returned by the DriverManager class’ method getConnection with the variables dbUrl, dbUser, and dbPassword passed to it. Lastly, the variable of the Connection instance is returned at the end. ***In the source code, there are lines that are commented out to get the database connection credentials from the system environment variables set in the IDE from the local machine. For the sake of the assignment, the credentials are hard-coded in the source code rather than deriving the values using the System class method getenv. In a production environment, this poses a security risk and is not standard practice.

- **TranslatorService:** This java file contains methods to retrieve each type of entity from the database based on user input. Each method takes in an object corresponding to the entity it is meant to retrieve (Ex. The retrieveSong method takes in a Song object). Each method checks the attributes of the passed in object to see the user specification, if any. If there is specification, the method appends the user input to a string named “where” and passes it and the table name to another method called getSVRS. The getSVRS method appends the passed in table name to the FROM clause and the passed in “where” string to the WHERE clause of a PreparedStatement, which gets executed and returns a ResultSet. The contents of each result in the ResultSet get stored in a LinkedHashMap. If the entity has multivalued attributes (Ex. Songs can have multiple artists, multiple releases, and belong to multiple labels.), then the getMVRS method is called for each matching result. The getMVRS method takes in parameters to be used in a PreparedStatement and loops through the returned ResultSet, appending each value to a string, separated by a semicolon. The getMVRS then returns an array containing the name of the multivalued attribute and the string containing all the appended semicolon-separated values. The contents of this array also get stored in the LinkedHashMap. Every LinkedHashMap gets stored in an ArrayList, which the retrieveXxx methods (Ex. retrieveSong, retrieveMusicArtist, etc.) return. If there is no user specification, then no query is made to the database, and the method returns an empty ArrayList.
- **AMDbController:** This java file contains the Get and Post methods used to add Model attributes to Spring Boot’s Model object. Adding an instance of each class in the ModelAttributes package to the Model allows the program to bring user input from the HTML page to the Java Application. Adding the ArrayList containing the LinkedHashMaps (results) to the Model allows the HTML file to print the database results to the user.

HTML File:

- ***home.html:***

Overview

This file is the main page used for inputting a query for the various categories of audio media. These categories include eleven entities related to music, podcasts, and audiobooks. The user can choose from multiple categories via a dropdown menu. The user can enter search criteria such as a song title, genre, ranges for the song's tempo/length, and whether or not a song is explicit. After the user enters their search criteria they can click "search" and start searching the database. Next, JavaScript updates the page to display a table of search results with the matching input.

HTML Structure

The HTML document follows a standard structure with a clear division between the head and body sections. The head section includes metadata, title, and links to external Bootstrap CSS for styling and icons. The body section is organized into several main parts: a header, search category dropdown menu button, search options tailored to different media types, and dynamically generated tables displaying search results.

Styling

Styling is achieved through embedded CSS within the head section and external Bootstrap frameworks. The design uses a dark theme, with dark backgrounds and light text to enhance readability and user experience. Specific classes and IDs are utilized for layout adjustments, responsiveness, and visibility control of elements based on user interactions.

Interactivity

The home.html webpage is enriched with interactive features handled by JavaScript. It includes:

- **Dynamic Search Options:** Based on the selected category from the dropdown, corresponding search fields are displayed. This is managed through the visibility toggling of div elements.
- **Input Validation:** JavaScript functions ensure that the input values for parameters like tempo and length are within acceptable ranges (only non-negative integer values and 'max' values must be greater than or equal to 'min' values) ensuring no erroneous search queries can be submitted.
- **Responsive Dropdown:** The dropdown menu updates its label based on the user's

selection and controls the display of associated input fields.

- **Alerts:** Alerts inform the user when no results are found or if input values are invalid, ensuring an intuitive user-friendly experience.

Search Functionality

The search functionality is the most critical aspect of the application. Upon selecting a category and specifying search criteria, the backend processes the queries against our AMDB database. The results are then displayed in tables formatted according to the type of media queried. Each table includes all relevant attributes such as IDs, names, and specific metadata according to the media type.

MySQL Database:

SQL Files:

- *create-AMDb.sql (Create/Delete Table statements)*
- *drop-AMDb.sql (drops the database if it already exists)*
- *insert-AMDb.sql (Insert and Indices statements)*

Music

- **Spotify Data:** We gathered data from a [pre-cleaned data set](#). This dataset provided the song id, artist name, album name, song name, duration, explicit boolean, tempo, genre, and extraneous attributes that were not required for our design. We used the dataset to get information for our Music_Release, Song, and Music_Artist entities. Unfortunately, the dataset did not provide record label information so we created dummy data based on a small set of popular record labels and randomly assigned them to each Artist and Music_Release.
- **Insert statements:** The insert statements for the music portion of our database contain entries for the entities: Record_Label, Music_Artist, Music_Release (album), and Song. We also created insert statements for the relationships between these entities (e.g. Record_label **releases** Music_Release, Music_Artist **records** Song).

Example music insert statements:

```
INSERT INTO Record_Label(Label_ID,Label_Name) VALUES (3,'Warner Music Group')
```

```
INSERT INTO Song(Song_ID,Song_Title,Genre,Tempo,Length,Explicit,Music_Release_ID)
VALUES (107,'Who (feat. BTS)', 'dance',142,180413,False,52)
```

```
INSERT INTO Music_Release(Release_ID,Release_Title) VALUES (52,'~how i'm feeling~')
```

```
INSERT INTO Music_Artist(Artist_ID,Artist_Name) VALUES (68,'BTS')
```

```
INSERT INTO Releases(Label_ID,Release_ID) VALUES (3,68)
```

```
INSERT INTO Signs(Label_ID,Artist_ID) VALUES (4,68)
```

```
INSERT INTO Creates(Artist_ID,Release_ID) VALUES (1,1)
```

```
INSERT INTO Records(Artist_ID,Song_ID) VALUES (1,1)
```

Podcast

- ***Apple Podcast Data:*** We found the [podcast dataset](#) off of GitHub that had data scraped from iTunes. It provided us with the podcast name, website url, itunes owner name, explicit boolean, and description for each entry. This dataset was not cleaned which meant we had to clean it using Python scripts. The podcast dataset did not contain podcast episode information so we created dummy data. The dummy data included a randomly generated number of episodes for each podcast as well as the podcast episode's title (e.g The podcast "The Soulful Room (Podcast)" would have 9 episodes and the title format would be "Episode 1 - The Soulful Room" with the episode number incrementing until episode 9).
 - ***Insert statements:*** The insert statements for the podcast entries include information about the Podcast_Host, Podcast, and Podcast_Episode entities.

Example podcast insert statements:

```
INSERT INTO Podcast(Podcast_ID,Podcast_Title,Url,Explicit,Description) VALUES  
(2,'8 Tracks with Dutch Bickell','http://dutchbickell.podbean.com',FALSE,'"8  
Tracks" off of an artist or bands newest/latest album release get covered with  
questions that take you behind the music and into the studio and deep into the  
creative process.')
```

```
INSERT INTO Hosts(Host_ID,Podcast_ID) VALUES (2,2)
```

```
INSERT INTO Podcast_Host(Host_ID,Host_Name) VALUES (2,'Dutch Bickell')
```

```
INSERT INTO  
Podcast_Episode(Podcast_Episode_ID,Podcast_Episode_Title,Podcast_ID) VALUES  
(10,'Episode 1 - 8 Tracks " with Dutch Bickell" (Podcast)',2)
```

Audiobook

- ***Audible Data:*** For the audiobook portion we found another [pre-cleaned dataset](#) based on Audible. This provided us with information for a book's title, author, narrator, length, release date. We were mainly able to use the information from this dataset. We were unable to obtain publishing information so we created dummy data similar to the Spotify

dataset where we gathered a small set of popular publishing companies and assigned them to each Audiobook.

- ***Insert statements:*** The insert statements include information about the Publisher, Audio_Book, Author, and Narrator entities.

Example audiobook insert statements:

```
INSERT INTO Publisher(Publisher_ID,Publisher_Name) VALUES (1,'Penguin Random House')
```

```
INSERT INTO Publishes(Publisher_ID,Book_ID) VALUES (1,2)
```

```
INSERT INTO Audio_Book(Book_ID,Book_Title,Length,Release_Date) VALUES (2,'The Burning Maze',788,'2018-01-05')
```

```
INSERT INTO Author(Author_ID,Author_Name) VALUES (2,'Rick Riordan')
```

```
INSERT INTO Writes(Author_ID,Book_ID) VALUES (2,2)
```

```
INSERT INTO Narrator(Narrator_ID,Narrator_Name) VALUES (2,'Robbie Daymond')
```

```
INSERT INTO Narrates(Narrator_ID,Book_ID) VALUES (2,2)
```

Indices:

Indices were created for each entity table using their primary key as the index and for each relationship table using each column as a separate index, in order to speed up querying of each entity's multivalued attributes.

Example index statements:

```
CREATE INDEX record_label_id_index ON Record_Label (Label_ID);
```

```
CREATE INDEX releases_label_id_index ON Releases (Label_ID);
```

```
CREATE INDEX podcast_host_id_index ON Podcast_Host (Host_ID);
```

```
CREATE INDEX hosts_host_id_index ON Hosts (Host_ID);
```

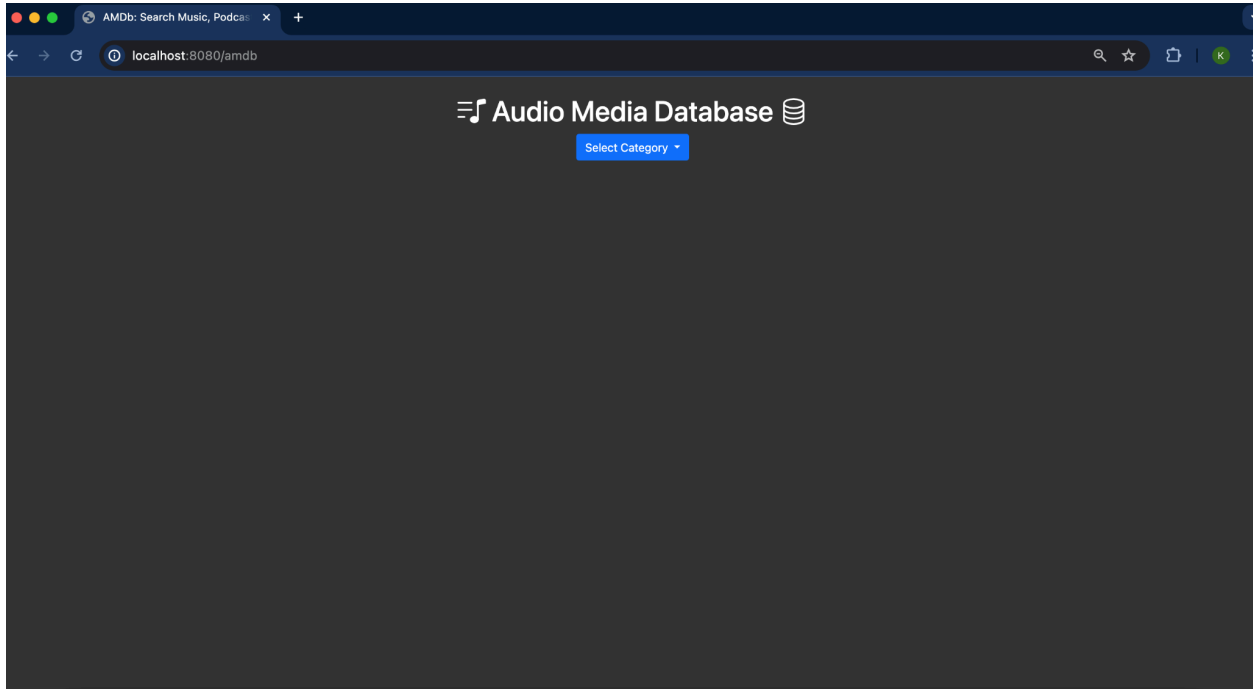
```
CREATE INDEX hosts_podcast_id_index ON Hosts (Podcast_ID);
```

```
CREATE INDEX publisher_id_index ON Publisher (Publisher_ID);
```

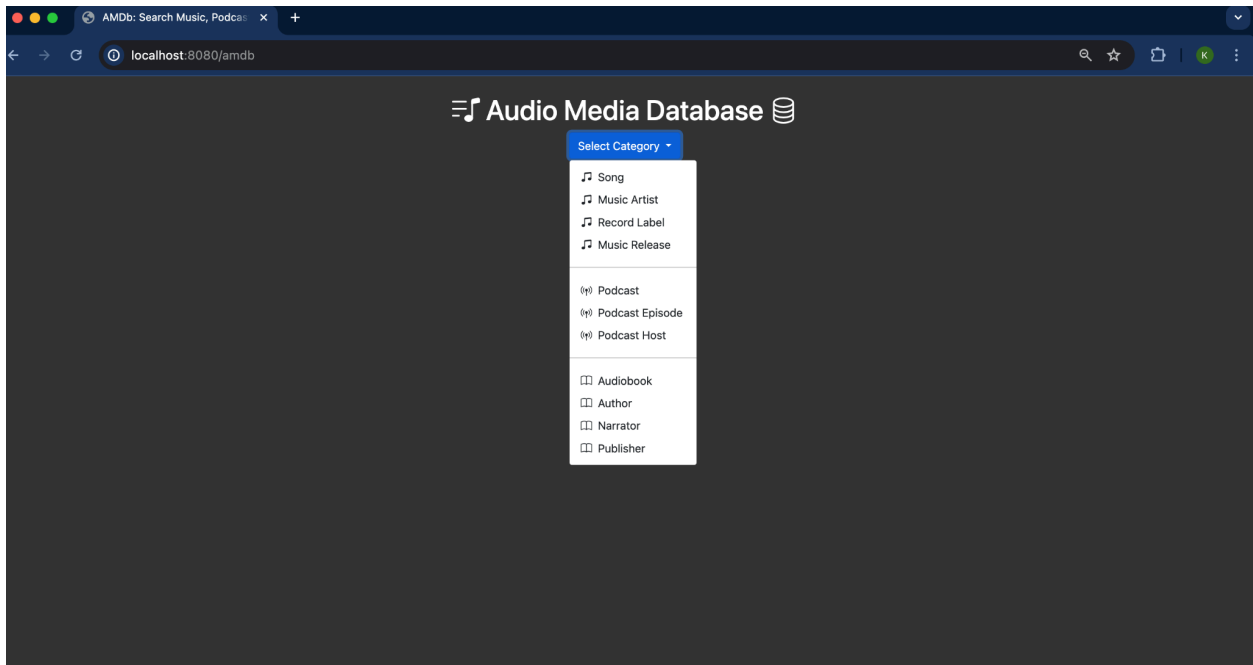
```
CREATE INDEX publishes_publisher_id_index ON Publishes (Publisher_ID);
```

Example System Run Screenshots

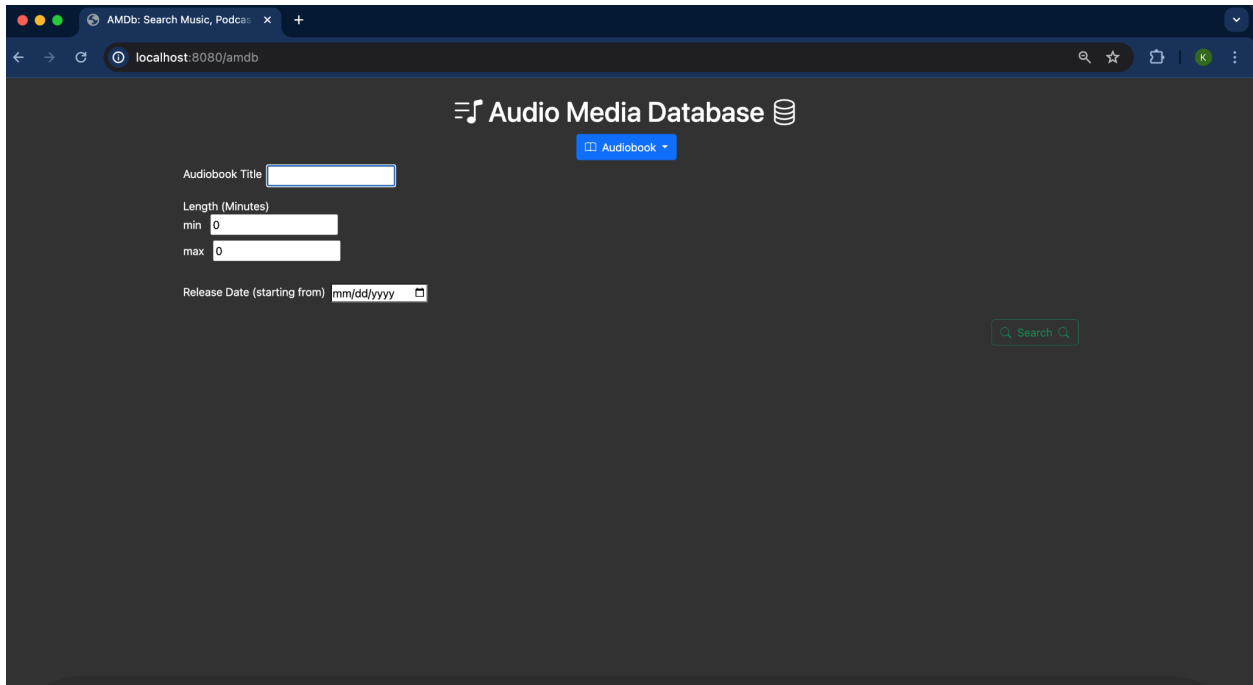
After entering `localhost:8080/amdb` into the web browser:



Opening the dropdown menu:



After selecting AudioBook from the dropdown menu:

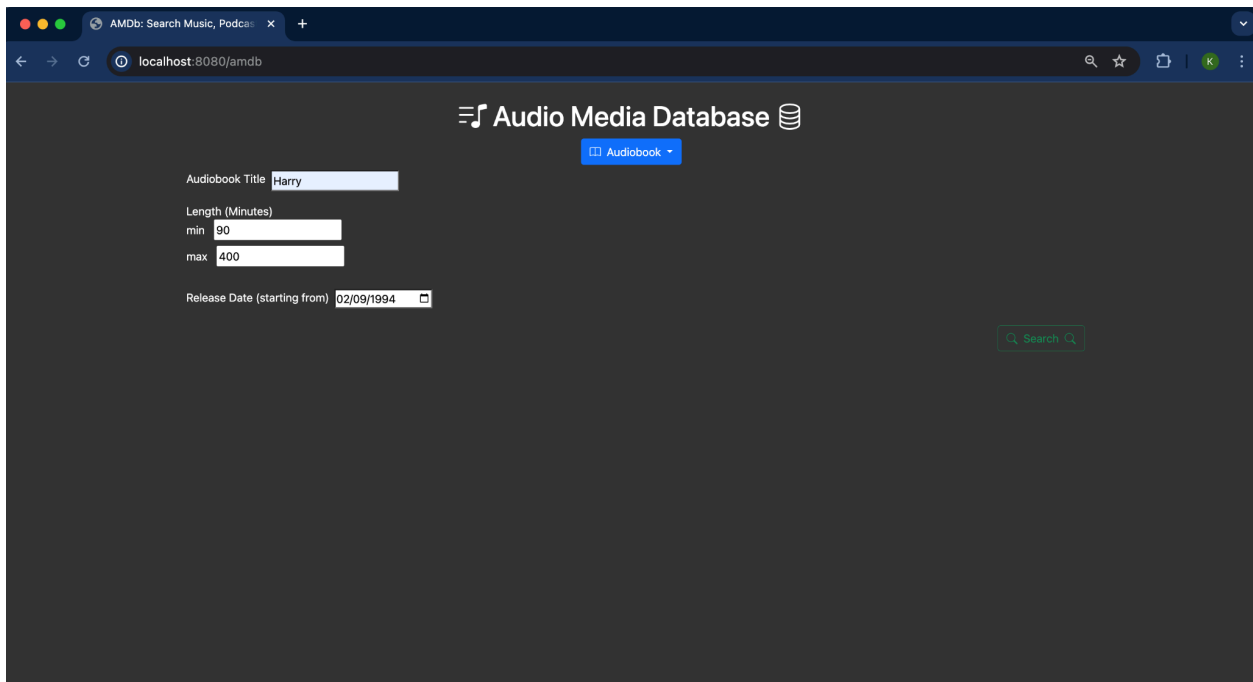


The screenshot shows a web browser window with the URL `localhost:8080/amdb`. The page title is "Audio Media Database" with a database icon. A blue dropdown menu is open, showing "AudioBook" selected. The search form includes the following fields:

- Audiobook Title:
- Length (Minutes):
 - min:
 - max:
- Release Date (starting from):

A green "Search" button is located on the right side of the form.

After typing input into the textboxes:



The screenshot shows the same web browser window, but the search form fields now contain input:

- Audiobook Title:
- Length (Minutes):
 - min:
 - max:
- Release Date (starting from):

The green "Search" button remains on the right side of the form.

After the user submits their input:

The screenshot shows a web browser at localhost:8080/amdb displaying the 'Audio Media Database' interface. The search criteria are: Audiobook Title 'Harry', Length (Minutes) min 90 and max 400, and Release Date (starting from) 02/09/1994. The search results table is as follows:

| Audiobook ID | Audiobook Title | Length | Release Date | Publisher | Author | Narrator |
|--------------|--|--------|--------------|-----------------------|--|----------------------------|
| 2870 | Harry Houdini | 155 | 2019-08-13 | Macmillan Publishers; | KjartanPoskitt; | PeteCross; |
| 7905 | Harry Versus the First 100 Days of School | 211 | 2021-06-29 | HarperCollins; | EmilyJenkins; | RebeccaSoler; |
| 14811 | Transforming Harry | 395 | 2018-07-31 | HarperCollins; | | EstherWane; ShaunGrindell; |
| 15610 | Harry Potter - The Ultimate Audiobook of Facts | 91 | 2020-04-21 | HarperCollins; | JackGoldstein; FrankieTaylor; HolgerWebling; | KentHarris; JackGoldstein; |
| 20547 | Harry S. Truman | 344 | 2008-02-09 | Hachette Livre; | RobertDallek; | WilliamDufres; |
| 20660 | Harry Haft: Survivor of Auschwitz, Challenger | 315 | 2020-02-11 | Scholastic; | AlanScottHaft; | PriceWaldman; |

Conclusion and Possible Improvements

Overall we were able to create a functional web application that met our goals. Our website is able to search through the database. However, we thought of some possible improvements to make the application better. One improvement would be adding more characteristics for the user to search by. For example we could add release dates for Song and Podcast entities, or adding a gender attribute Music_Artist, Podcast_Host, Author, and Narrator entities. Another improvement would include allowing the user to choose between exact and partial String matching. For exact matching the user would type in “pop” and songs that only have “pop” as the genre would be displayed. If a user selected partial matching they could type “pop” into the search criteria and the application would return results with “pop”, “cantopop”, “k-pop”, etc. Currently, only partial matching is implemented. Finally, it would be nice to use real data instead of dummy data for some of the entries in the database. The datasets we found did not provide information for the Record_Label, Podcast_Episode, and Book_Publisher entities so we had to create randomly generated dummy data for them. In the future it would be nice to implement accurate data to provide authenticity and credibility to our application.