Database Systems

BLG 317E - 13593

Term Project Report

Elif Feyza Güler - 150200041
gulere20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 23.12.2024

1. Introduction

For my term project, I built a web application using the Chinook database. I chose this database because it has a well-structured design for managing a music store, with data about tracks, albums, artists, playlists, customers and employees. I used React for the frontend and Node.js for the backend to create a connection between the user interface and the database. My goal was to make an easy-to-use platform where users can explore and manage music-related data. Through this project, I learned more about full-stack development, creating APIs, and connecting the frontend with the backend.

The application includes features for listing tracks, artists, albums, playlists, employees, customers, allowing users to explore music-related data easily. I also developed a multifunctional admin system where users can edit records in all these tables, add new entries, or delete existing ones. I focused on keeping the design simple yet visually appealing, ensuring the interface is both easy to navigate and pleasant to use.

I didn’t have a team, therefore I did all the work by myself.

2. Implementation Details

2.1. Backend

For the backend of my project, I used a controller-repository-router structure to organize and implement all endpoints efficiently. The router handles incoming HTTP requests and directs them to the appropriate controller. The controllers manage the application logic, processing requests and preparing responses. To keep the code modular and maintainable, I used a repository layer to interact with the database, ensuring a clear separation of concerns. This structure allowed me to handle CRUD operations for all tables systematically, making it easier to manage and scale the backend as needed.

I used different controller and repository files for different tables. Controllers and repositories of tracks, albums, artists etc. is separate from each other.

2.1.0.1. Router

In my project, I used routers to keep the code organized and modular. For example, the router for the tracks feature includes endpoints like GET /artists to retrieve all artist, POST /artists/add to add a new artist, PUT /artists/update/:id to update an artist, and DELETE /artists/delete/:id to delete an artist. Each of these routes forwards the request to the corresponding controller function, ensuring a clear separation of responsibilities.

```
const express = require("express");
const router = express.Router();
const {
  fetchArtists,
  fetchArtistById,
  addArtist,
  updateArtist,
  deleteArtist,
} = require("../Controllers/artistController");

router.get('/artists', fetchArtists);
router.post('/tracks/add', addArtist);
router.put('/artists/update/:id', updateArtist);
router.delete('/artists/delete/:id', deleteArtist);

module.exports = router;
```

Figure 2.1: Router Example

2.1.0.2. Controllers

In my project the controller serves as the central part of the backend logic, acting as a bridge between the router and the repository. It receives requests from the router, processes them, and prepares responses to be sent back to the client. The main

responsibility of the controller is to handle the application logic, ensuring that each request is processed correctly according to the functionality required.

For example *addAlbum* function is a controller method responsible for handling the logic of adding a new album to the database. It begins by extracting the title and artist_id from the request body sent by the client. The function performs several validation checks to ensure data integrity. First, it verifies that the album title is provided and not empty; if it is missing, a 400 Bad Request response is returned with an error message. Next, it checks if the artist_id is present; if not, a default value of 276 is assigned, representing "No Artist" in the database. The function then validates that the artist_id is a valid number, returning an error if it is not. Once the inputs are validated, the addAlbumToDatabase function from repository is called to insert the new album into the database. If the operation is successful, a 201 Created response is sent back to the client, including a success message and the newly added album. In case of any errors during this process, the function forwards the error to the next middleware for handling.

```
const addAlbum = async (req, res, next) => {
  try {
    let { title, artist_id } = req.body;

    if (!title || title.trim() === "") {
      return res.status(400).json({ error: "Album title is required" });
    }
    if (!artist_id || artist_id == undefined) {
      artist_id = 276; //No artist value
    }

    if (isNaN(parseInt(artist_id))) {
      return res.status(400).json({ error: "Artist ID is not valid" });
    }
    const newAlbum = await addAlbumToDatabase({ title, artist_id });

    res.status(201).json({
      message: "Album added successfully",
      album: newAlbum,
    });
  } catch (err) {
    next(err);
  }
};
```

Figure 2.2: Controller Example

2.1.0.3. Repositories

In the repository layer, the main responsibility is to handle direct interactions with the database, ensuring a clear separation between database operations and business logic. For example, the repository function for adding an album, addAlbumToDatabase, receives the validated title and artist_id from the controller and constructs a query to insert this data into the database. Once the album is successfully added, the repository returns the newly created record or relevant information to the controller. This layer abstracts the database logic from the rest of the application, making it easier to maintain and modify database operations without affecting other parts of the codebase.

```
const addAlbumToDatabase = async (albumData) => {
  const { title, artist_id } = albumData;

  const query = `
    INSERT INTO album (title, artist_id)
    VALUES ($1, $2)
    RETURNING *;
  `;

  const values = [title, artist_id];

  const { rows } = await pool.query(query, values);
  return rows[0];
};
```

Figure 2.3: Repository Function Example

2.1.0.4. All CRUD operations and Query Explanations

Tracks

- **Fetch All Tracks (getAllTracks):** This query is to retrieve a detailed list of all tracks in the database. It joins the track table with the genre, album, and artist tables to provide additional information for each track. The query selects details such as the track ID, track name, album ID, album title, composer, artist name, genre name, track duration, and the unit price of the track. The results are sorted alphabetically by the track name (ORDER BY track.name ASC).

- **Fetch Tracks by Artist ID (getTracksByArtistId)**: This query retrieves all tracks associated with a specific artist using a ***nested query***. The main query joins the track, album, genre, and artist tables to get detailed information about each track, including the track ID, name, album ID, album title, composer, genre, duration, and unit price. The WHERE clause uses a subquery to filter albums belongs to the given artist ID (artistId). This ensures that only tracks from albums linked to the specified artist are included in the results. The query uses parameterization ($1) to safely insert the artist ID, *protecting against SQL injection*.

- **Fetch Tracks by Album ID (getTracksByAlbumId)**: This query retrieves all tracks within the given album. Like the previous query, it joins the track, album, genre, and artist tables to get information such as the track ID, track name, album title, composer, genre, duration, and price. The filtering is done by matching the album ID provided by the user (track.album_id = $1).

- **Add a New Track (addTrackFromDatabase)**: This query handles the insertion of a new track into the track table. It accepts data for fields: track name, album ID, media type ID, genre ID, composer, duration (milliseconds), and unit price. These values are dynamically inserted into the query using placeholders ($1 through 7$), ensuring the operation is secure and prevents SQL injection. The bytes field,

which may represents file size, is set to NULL. The query concludes by returning the newly added track record, making it easy to confirm that the operation was successful.

- **Update a Track (updateTrackFromDatabase):** This query is used to update one or more fields of an existing track in the database. It dynamically constructs the SET clause based on the fields provided in the input. Each field to be updated is paired with a corresponding placeholder (e.g., $1,2$) for the new value. The query ensures that at least one field is provided for updating; otherwise, it throws an error. The track ID is included as the last parameter to identify the specific record to update. The query returns the updated track record, allowing verification of the changes.
- **Delete a Track (deleteTrackFromDatabase):** This query deletes a specific track from the database by its ID. The track_id is used as a filter, and its value is safely inserted using a placeholder ($1). The query returns the deleted track record. This query ensures a secure and efficient way to manage the removal of tracks from the database.

Albums

- **Fetch All Albums (getAllAlbums):** This query retrieves a list of all albums from the database, including each album's details and the corresponding artist's name. It joins the album table with the artist table on the artist_id column and orders the results alphabetically by album title.
- **Fetch Albums by Artist ID (getAlbumsByArtistId):** This query retrieves all albums created by a specific artist. It selects details like the album ID, album title, artist ID, and artist name by joining the album and artist tables. The results are filtered by the provided artist ID and ordered by album title.
- **Fetch Album by Album ID (getAlbumByAlbumId):** This query retrieves the details of a single album based on its album ID. The WHERE clause filters the album using the provided ID, and the result is returned directly from the album table.
- **Add a New Album (addAlbumToDatabase):** This query inserts a new album into the album table. It accepts the album's title and artist ID as inputs, which are dynamically inserted into the query using placeholders. The query returns the newly created album record.
- **Update an Existing Album (updateAlbumInDatabase)**: This query updates specific fields of an album in the album table. The fields to update are dynamically constructed based on the input, with placeholders for the new values. The album

ID is used to identify the record to update, and the query returns the updated album.

- **Delete an Album (deleteAlbumFromDatabase)**: This query removes an album from the database based on its ID. The album_id is passed as a parameter, and the query returns the deleted album's details.

Artists

- **Fetch All Artists (getAllArtists)**: This query retrieves a list of all artists from the artist table. The results are ordered alphabetically by the artist's name.
- **Fetch Artist by ID (getArtistById):** This query retrieves a specific artist's details based on their artist ID. The WHERE clause filters the results by the provided ID, and the result is ordered by the artist's name.
- **Add a New Artist (addArtistToDatabase):** This query inserts a new artist into the artist table. It takes the artist's name as input, dynamically inserts it using a placeholder, and returns the newly created artist record.
- **Update an Artist (updateArtistInDatabase):** This query updates specified fields of an artist's record in the artist table. The fields to update are dynamically constructed based on the input, using placeholders for the new values. The artist ID is used to identify the record to update, and the updated artist is returned.
- **Delete an Artist (deleteArtistFromDatabase):** This query deletes an artist from the database based on their artist ID. The ID is passed as a parameter, and the query returns the deleted artist's record.

Playlists

- **Fetch All Playlists (getAllPlaylists):** This query retrieves all playlists from the playlist table.
- **Fetch Playlist by ID (getPlaylistsById):** This query retrieves a specific playlist's details based on its ID. The WHERE clause filters the playlist using the provided playlist_id.
- **Add a New Playlist (addPlaylistInDatabase):** This query inserts a new playlist into the playlist table. The playlist name is dynamically inserted using a placeholder, and the newly created playlist record is returned.
- **Update a Playlist (updatePlaylistInDatabase):** This query updates the name of an existing playlist in the playlist table. The playlist ID is used to identify the record, and the new name is passed dynamically. The query returns the updated playlist record.

- **Delete a Playlist (deletePlaylistFromDatabase):**This query removes a playlist and its associated tracks from the database(from playlist_track table). It uses a ***transaction*** to first delete all records in the playlist_track table linked to the playlist ID, then deletes the playlist itself from the playlist table. If any error occurs, the transaction is rolled back to maintain data integrity. The deleted playlist record is returned.
- **Add Tracks to a Playlist (addTracksToPlaylistInDatabase):**This query adds one or more tracks to a specific playlist by inserting entries into the playlist_track table. For each track, the playlist ID and track IDs are dynamically inserted, and the newly added records are returned.
- **Fetch Tracks in a Playlist (getTracksInPlaylistFromDatabase):** This query retrieves all tracks associated with a specific playlist. It joins the playlist_track, track, album, and artist tables to provide detailed information about each track, including the track name, album title, artist name, and track duration.
- **Remove a Track from a Playlist (removeTrackFromPlaylistInDatabase):** This query deletes a specific track from a playlist by removing the corresponding record in the playlist_track table. The playlist ID and track ID are used to identify the record, and the deleted entry is returned.

Customers

- **Fetch All Customers (getAllCustomersFromDatabase):** This query retrieves all customers from the customer table. The results are ordered alphabetically by the customers' first names.
- **Add a New Customer (addCustomerToDatabase):** This query inserts a new customer into the customer table. It dynamically accepts customer details such as first name, last name, address, and more. These values are inserted using placeholders to prevent SQL injection. The query returns the newly created customer record.
- **Update a Customer (updateCustomerInDatabase):** This query updates specific fields of an existing customer in the customer table. It dynamically constructs the SET clause based on the fields provided in the input, using placeholders for the updated values. The customer ID is used to identify the record, and the updated customer record is returned. If no fields are provided for updating or if an error occurs, appropriate error handling is triggered.
- **Delete a Customer (deleteCustomerFromDatabase):** This query removes a customer from the database by their customer ID. The ID is passed as a parameter, and the query returns the deleted customer record.

Employees

- **Fetch All Employees (getAllEmployeesFromDatabase):** This query retrieves all employees from the employee table, ordered alphabetically by their first names.
- **Add a New Employee (addEmployeeToDatabase):** This query inserts a new employee into the employee table. It accepts details about the employee, such as their first name, last name, title, birth date, hire date, and contact information. The values are inserted securely using placeholders (1,2, etc.) to prevent SQL injection. The query returns the newly added employee record.
- **Update an Existing Employee (updateEmployeeInDatabase):** This query updates specific fields of an employee's record in the employee table. It dynamically constructs the SET clause based on the input fields and values. The employee ID is used to identify the record to be updated, and the updated employee record is returned. If no fields are provided or the employee ID is invalid, appropriate error handling is triggered.
- **Delete an Employee (deleteEmployeeFromDatabase):** This query removes an employee from the database using their employee ID. The ID is passed as a parameter. The query returns the deleted employee's record. If the employee does not exist or an error occurs, an appropriate error message is logged, and an error is thrown.

Admin Statistics

- The **dashboardStatsRepository** provides a collection of functions to retrieve counts for entities in the database and enables quick access to statistics for a dashboard. Each function executes a COUNT(*) query on a specific table (track, artist, album, customer, playlist, employee) to calculate the total number of records in that table. The result is parsed as an integer and returned.

2.2. Frontend

I developed the frontend of the project using React with JavaScript, aiming to create a dynamic user interface. Each key feature, such as managing tracks, artists, albums, playlists, and customers, was implemented as a separate component to simplify development and ensure maintainability.

I focused on keeping the design clean and visually appealing by using TailwindCSS.

To manage navigation across the application, I integrated React Router. I defined routes for different pages, such as:

- /tracks: To display a list of tracks.
- /artists: To list all artists.

- /albums: To show album details.
- /playlists and /customers: To provide access to playlist and customer management.
- /dashboard: To show key statistics and an overview of the database.
- and many more.

I divided the user interface into smaller components. This made the application easier to develop and debug. Each component was designed to handle specific functionality, such as fetching data from the backend, rendering the content, and interacting with the user.

I connected the frontend to the backend by using Axios to fetch and update data through the RESTful API. Each feature, such as adding or editing records, was implemented by calling the respective backend endpoints.

I implemented a home page and side bar to create a user-friendly experience.

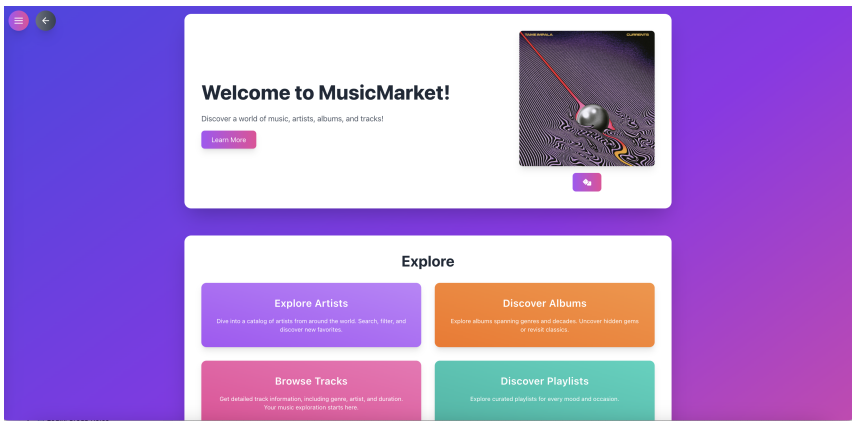


Figure 2.4: Home Page

I implemented a multifunctional **admin dashboard** to allow users to view statistics like track, artist, and album counts. Also admin dashboard allows all CRUD operation on tracks, artists, albums, playlists, customers and employees. Each of these has its own page for the operations. For add and update operation forms I used widgets within page.

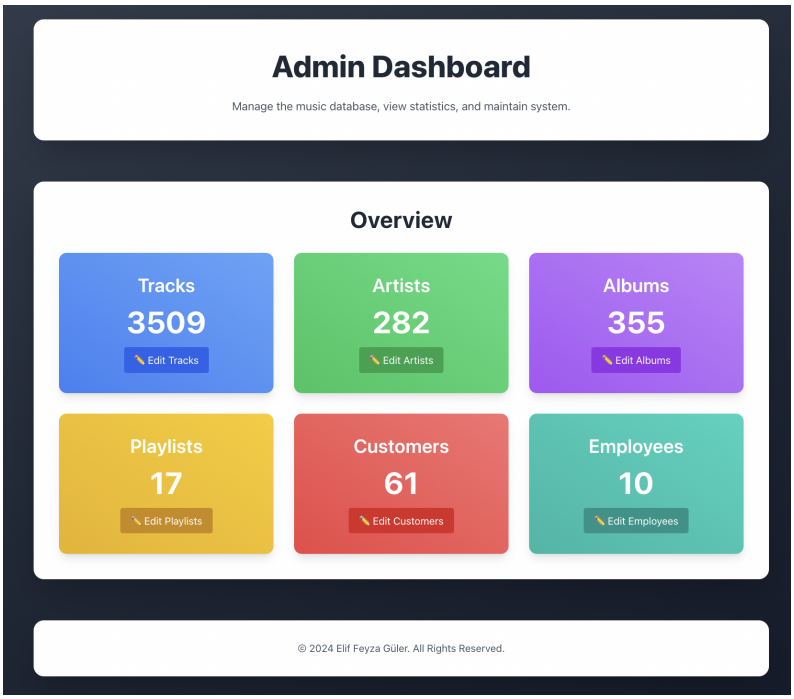


Figure 2.5: Admin Dashboard

Also admin dashboard allows all CRUD operation on tracks, artists, albums, playlists, customers and employees. Each of these has its own page for the operations. For add and update operation forms I used widgets within page. I also added search bar and pagination to each listing page in application. I added an example admin page for managing tracks. I provided other page's screenshots at the end of the report.

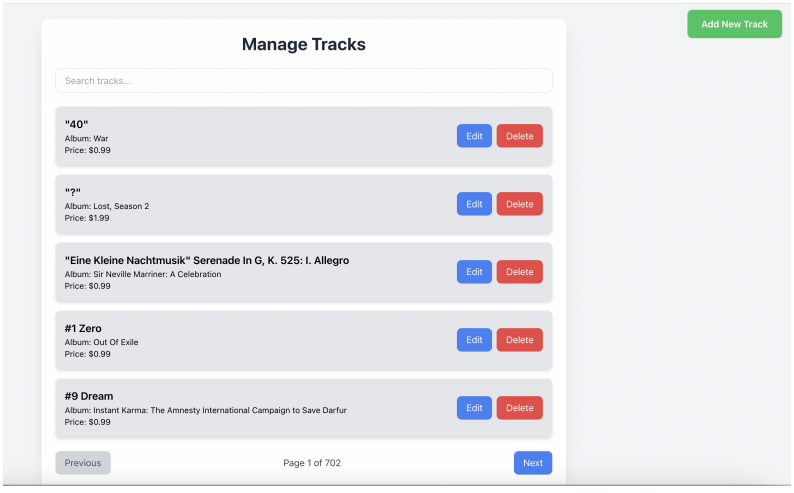


Figure 2.6: Manage Tracks Page

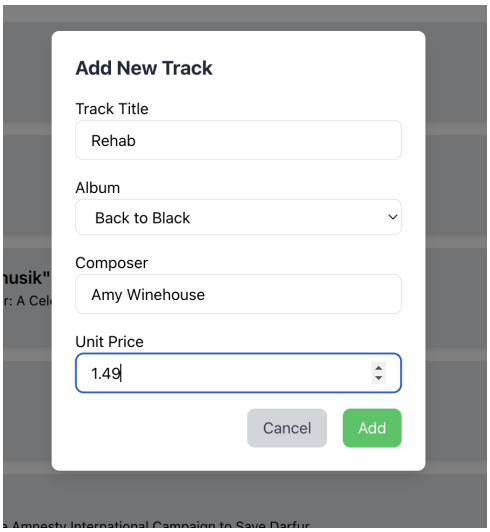


Figure 2.7: Add Track Widget

I also added an admin login page (with static username and password) to make the project closer to a real life example.

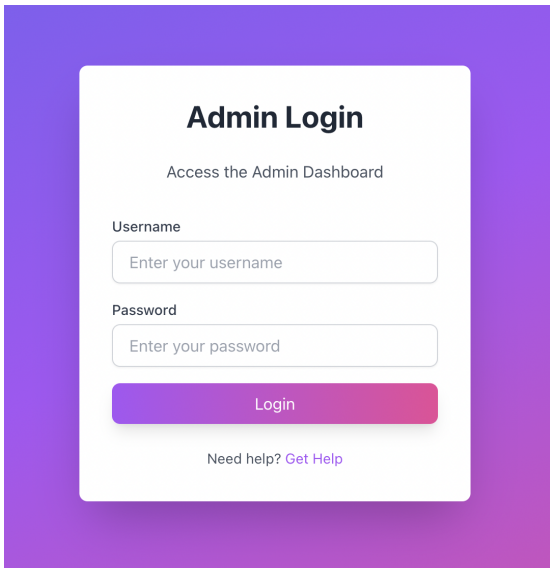


Figure 2.8: Admin Login Page

I implemented playlist page in a colorful way using cards.

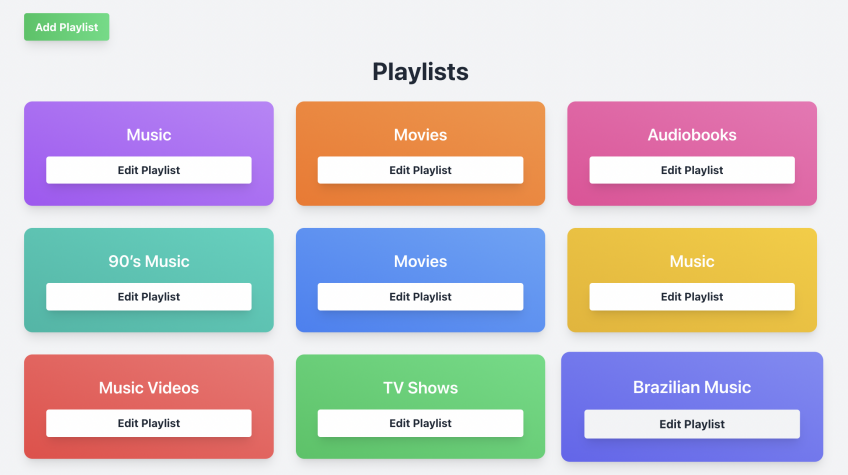
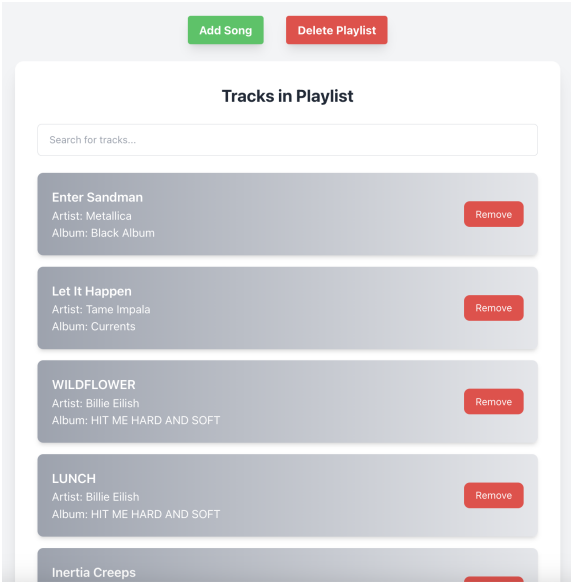
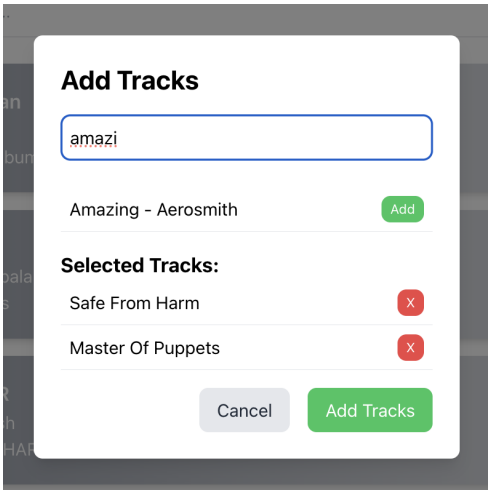


Figure 2.9: Playlists



((a)) Manage Playlist



((b)) Add Track To Playlist

Other pages are designed in the same way, easy to use and visually appealing.

3. ERD Diagram and Other Pages of Project

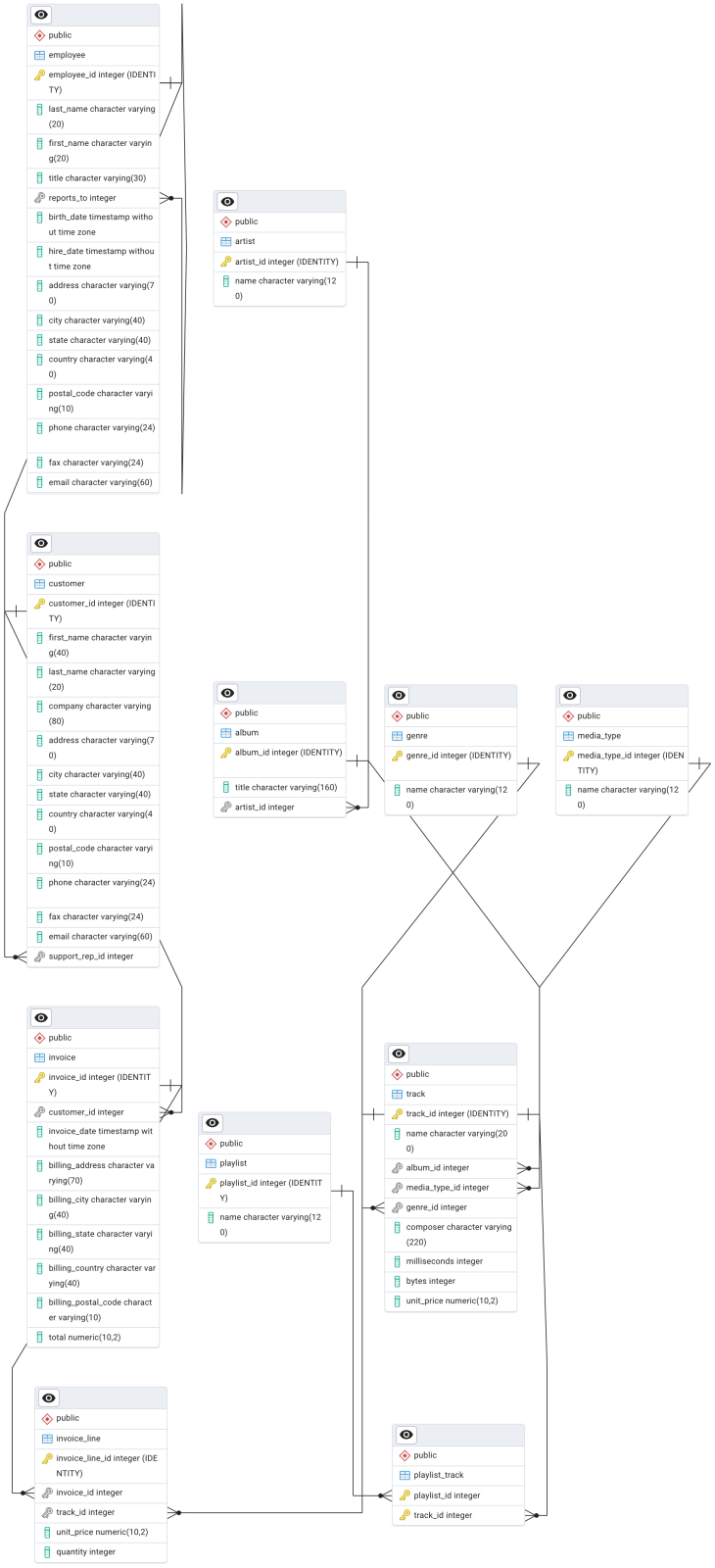


Figure 3.1: ERD Diagram

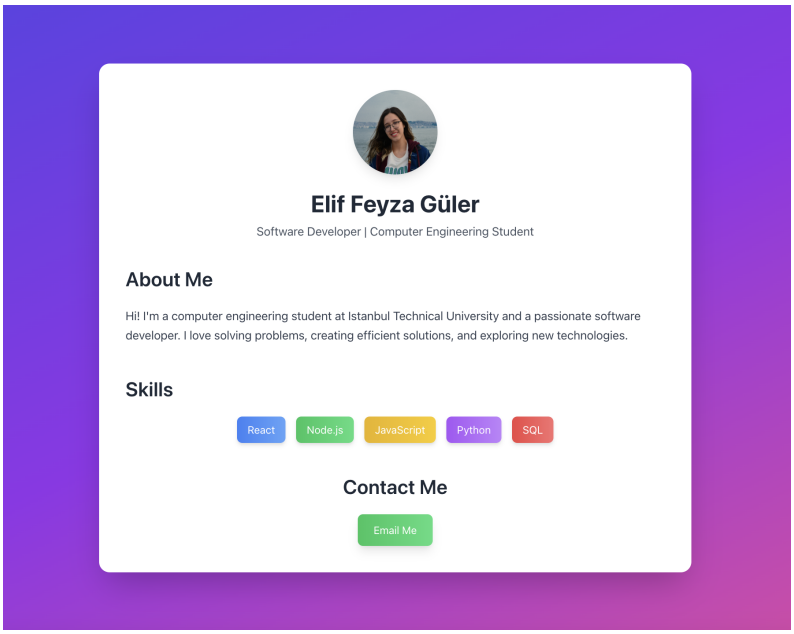


Figure 3.2: About Me Page

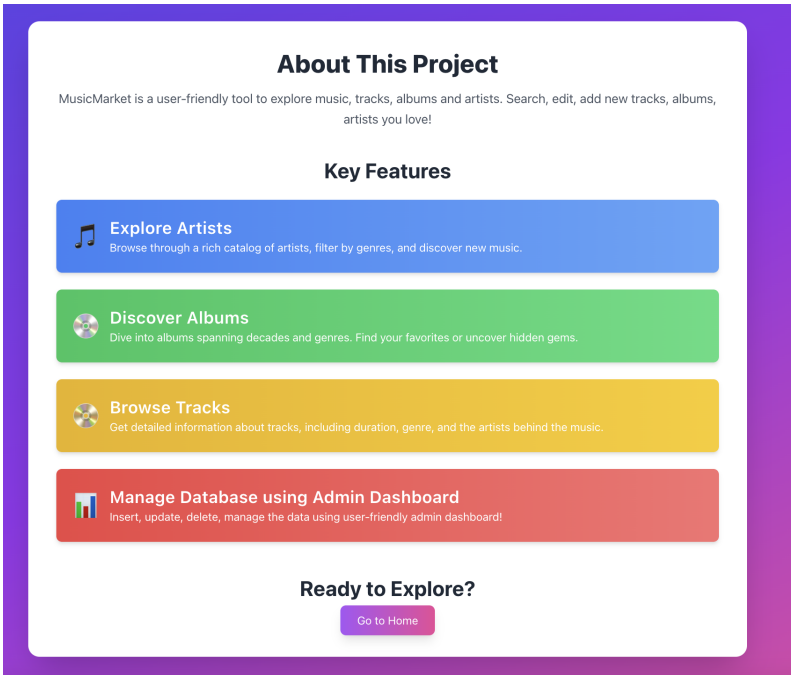


Figure 3.3: About Project Page

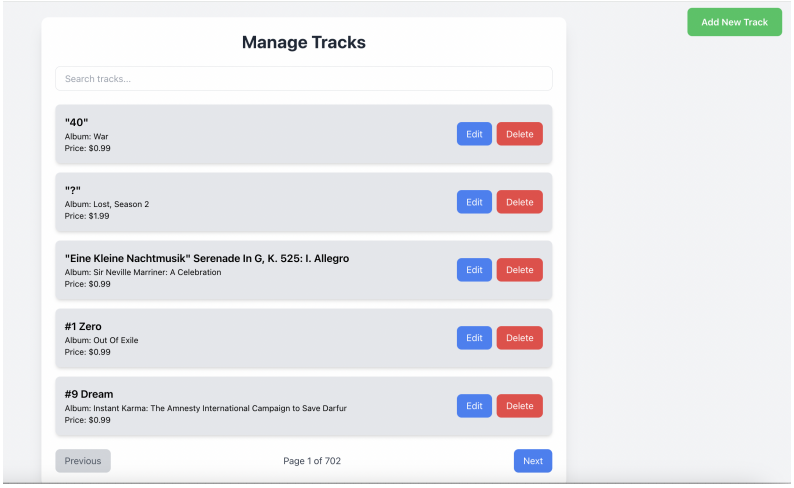


Figure 3.4: Manage Tracks Page

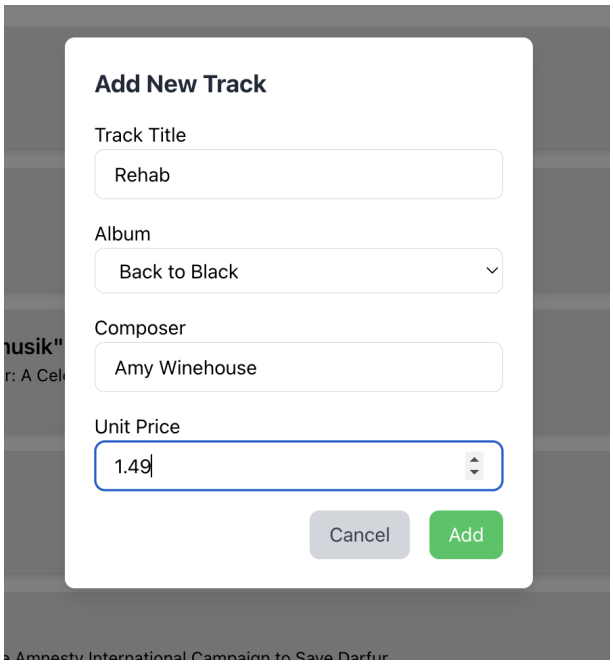


Figure 3.5: Add Track Widget

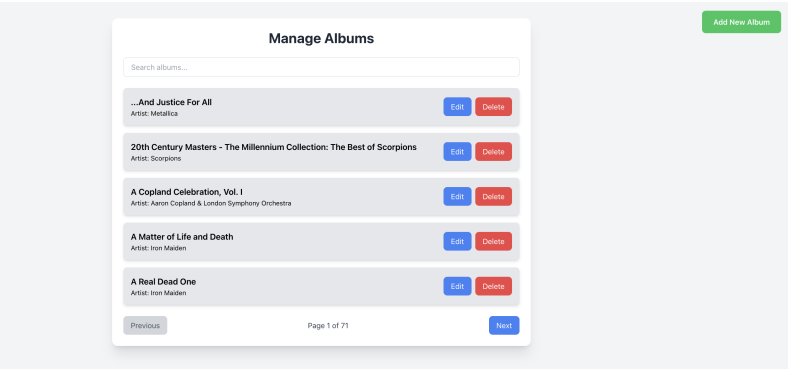


Figure 3.6: Manage Albums Page

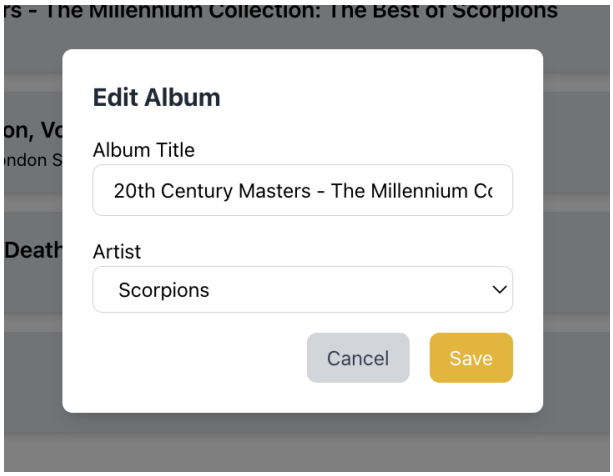


Figure 3.7: Edit Album Widget

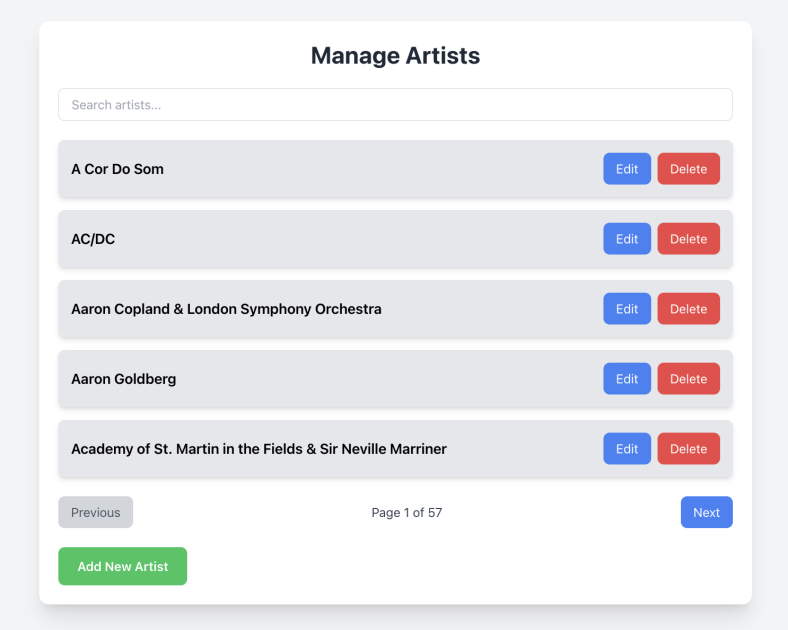


Figure 3.8: Manage Artist Page

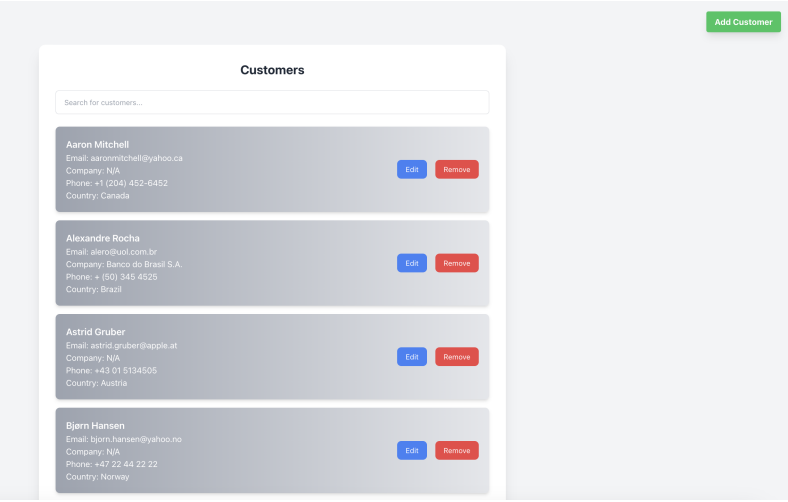


Figure 3.9: Manage Customers Page

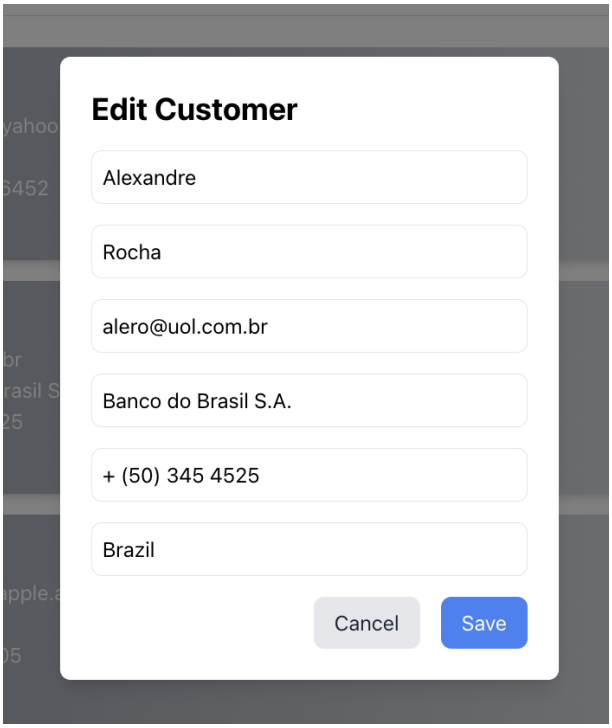


Figure 3.10: Edit Customer Widget

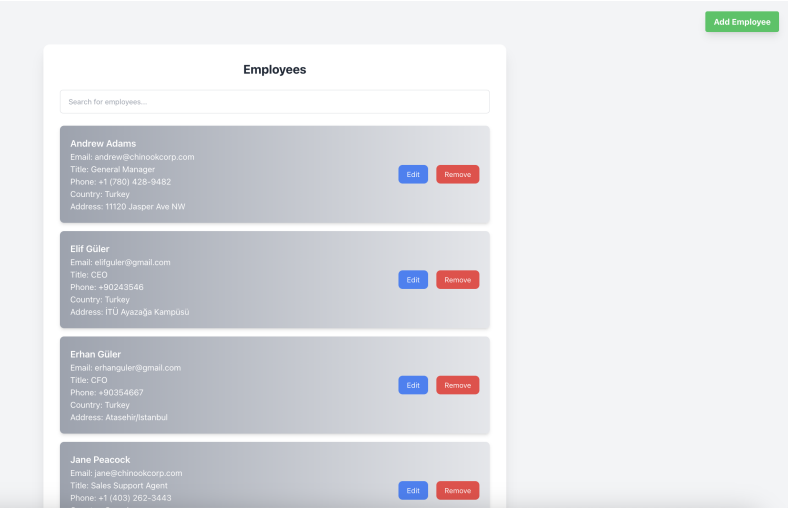


Figure 3.11: Manage Employee Page

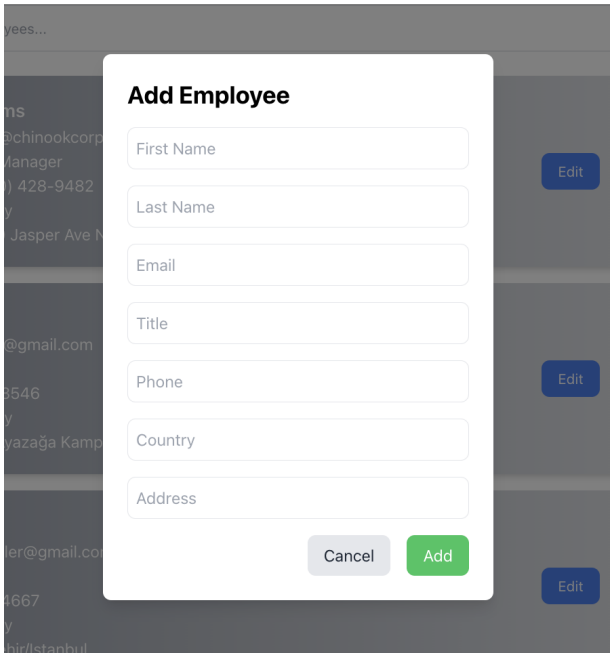


Figure 3.12: Add Employee Widget

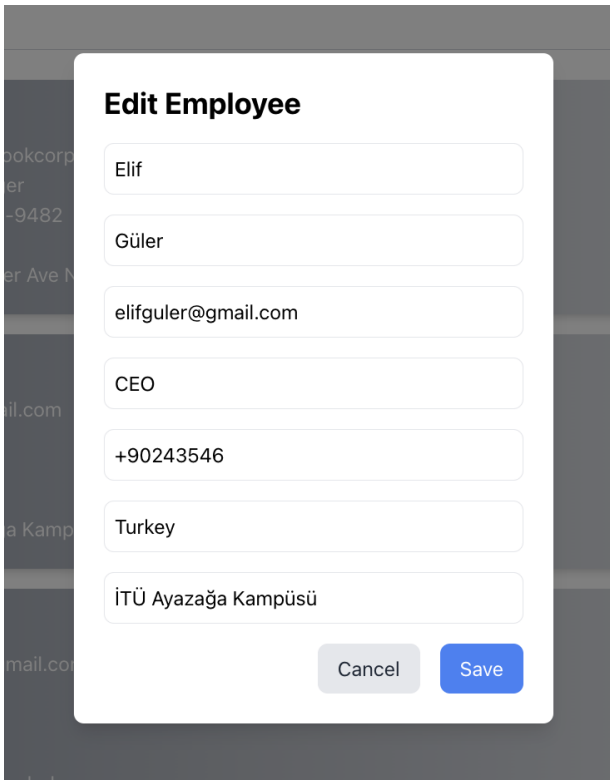


Figure 3.13: Edit Employee Widget