



AUGUST 5, 2021

ALBUM COLLECTION USING DISCOGS API

CST 336: INTERNET PROGRAMMING

LARRY CHIEM, IAN ROWE, RAYMOND SHUM, NICHOLAS STANKOVICH



Table of Contents

[1.0] Overview	2
[1.1] Website Title and Description	2
Title	2
Description	2
[2.0] Task Distribution.....	2
[3.0] Changes from the Original Design	2
[4.0] Database Schema	3
[4.1] Design	3
[4.2] Implementation	4
User	4
Wishlist & Collection	4
Review	4
Album	4
[5.0] Flowchart: User Experience	5
[6.0] Rubric Requirements	6
[6.1] Project allows user interaction with at least three different types of form elements (text box, select, radio, checkbox, etc)	6
[6.2] Project uses at least three database tables with at least 10 fields (combined).....	7
[6.3] Project uses Web Storage or Sessions.....	7
[6.4] Project allow users to update existing records in the database, in a friendly way (data is pre-filled)	7
[6.5] Project allow users to add records to the database.....	8
[6.6] Project must have at least 100 lines of JavaScript/jQuery code	8
[6.7] Project includes at least three local or external Web APIs. As part of your submission, please explain where the Fetch calls are.	8
What are fetch calls?	9
[6.8] Project has a nice, professional, and consistent design, free of typos. Uses at least 50 CSS properties or Bootstrap.	9
[7.0] Open Issues & Known Bugs	9
[8.0] Future Implementation (Conclusion)	10
[9.0] Screenshots	10

[1.0] Overview

The purpose of this paper is to meet the requirements outlined in the rubric and serve as documentation for Team innovatree's final project for CST 336. We intend to use this paper as part of our portfolio and for future interviews, so we have included more information than is necessary. Please feel free to use both the table of contents (if using word) or bookmarks (if using Adobe Reader) to easily navigate the document.

Following the guidelines, the following topics are covered in this paper: Title, Description, Task Distribution, Changes from the original design, Database Schema, Screenshots of the finished product. As part of the requirements, we will also describe fetch calls in this paper and on the Canvas submission post. To assist with grading, we have included a section describing how we have fulfilled the rubric requirements.

[1.1] Website Title and Description

Title: Album Collection using Discogs API

Description: This website allows users to find albums, add them to their collection or wishlist and read or write global (for the website) reviews. Upon reaching the landing page, users will be prompted to sign up if they don't already have an account. Otherwise, they will log in. Once the user authenticates, they proceed to their dashboard, where they can view their wishlist, collection and reviews that they've written. From the dashboard, they can search for new albums to review or associate with their accounts.

[2.0] Task Distribution

Based on feedback, we decided that each member have a role in implementation of all aspects of the website: front-end, middleware and database. This is to ensure that all members gain full stack experience while working on the project. As such, all members have contributed equally to all aspects of implementation for all features. We are all learning how to collaborate using GitHub and to properly structure our project, we decided to have a single member, the team coordinator, issue pull requests to avoid merge conflicts when possible.

[3.0] Changes from the Original Design

Originally, our group had planned on implementing a hotel booking system. However, after further conversation, we determined that our current project was both more interesting and relevant to our lives outside the course. All of us maintain record collections.

In our current project, we had planned for the website to allow a user to track the total value of their collection as well as the value of single albums. However, we encountered the below issue that we were unable to resolve during the time we had budgeted for project work.

There does not seem to be a way to retrieve price information from the Discogs API for multiple releases in a single request. At the same time, price seems to be associated with the record for each individual release. If we wanted to display the current value of an individual's record collection, we would have to make fetch requests equal to the number of albums currently in their collection. Potentially, a single user with a large enough collection would exhaust our pool of available requests from the Discogs API.

One solution would be to create a new weak entity called “price” associated with our “album” table. This table would contain the album id, album price and a CURRENT_TIMESTAMP() marking the record’s entry or modification. We could either update each record after a certain amount of time has passed or all records on a scheduled basis. This would allow us to distribute fetch requests in a way that would not deny availability to our users. However, users would not see the current total value of their album collection and would see the last updated value instead.

In the end, we decided to display the price of individual albums on each album page because that is already part of a fetch request that must be made.

[4.0] Database Schema

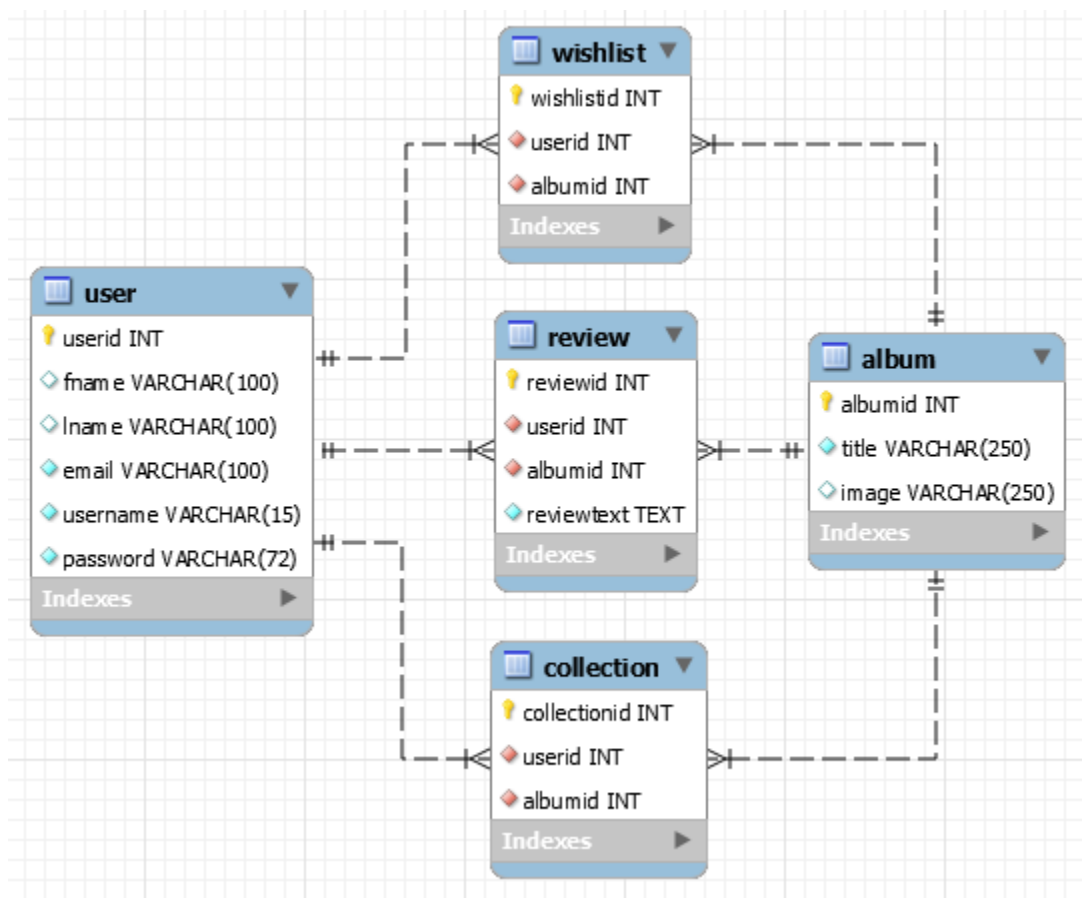


Figure 4.1 - Database Schema

[4.1] Design

Our database schema consists of 5 tables: **user**, **wishlist**, **collection** and **album**. There are two parent entities: **user** and **album**. The three child entities (**wishlist**, **collection** and **review**) each hold foreign keys pointing to the primary key of their parent tables. Logically, each child table establishes a specific relationship shared by each instance of **user** and **album**. For example, this album is in this user’s wishlist or this user wrote a review about this album.

The user table has 1:Many relationships with the wishlist, collection and review tables. This is because there can be many instances of each entity associated with a single user. Every album in a user's wishlist or collection is represented by an additional record in these tables. The same applies for each record that a user has reviewed.

The album table also has 1:Many relationships with its child entities. Each album can be present in the wishlist or collection of many users. They can also be reviewed by many different users.

[4.2] Implementation

User

A user record is created once a user creates an account for the website. We included several fields (such as name and email) to track identifying information for the purpose of account recovery. However, more columns can be added (such as gender and address) for the purpose of billing or analytics. For the current implementation, a user must provide a unique username, unique email, and non-unique password. The Node server will store the userid as a session variable and retrieve it to use as a parameter for MySQL queries.

Wishlist & Collection

Records are created in these tables when a user browses to an album page and selects the option to add it to either their wishlist or collection. They are also removed when the corresponding action is selected. Before adding a record either table, the Node server will check to see if a record containing the corresponding albumid is present in the album table. If not, it adds the album along with the image URL and title. This is due to a limitation of the Discog API mentioned in Section 2.0.

Records are retrieved from this table when a user browses their wishlist or collection tabs within their respective user dashboard. Either the wishlist or collection table is joined to the album table and the resulting information is displayed to the user.

Review

Records are added to this table when a user selects the option to write a review from the review page. They are deleted if the user selects the option to remove a review. A user can have one review for an album that is already present in their collection. They can modify or delete this review.

An alternative implementation would be to have the albumid of review point to the albumid of collection. However, we would have to join three tables (review, collection, and album) rather than the current two (review and album). We implemented client-side validation using a local API to display the correct options to the user.

Album

Records are added to this table when a user adds an item to their collection or wishlist. However, they are not removed. Records are retrieved from this table and displayed to the user when they view their wishlist, collection or review tabs inside their user dashboard. This table is meant to reduce the number of fetch requests made to the Discogs API by caching relevant data.

Originally, we intended to make use of Discogs API to retrieve the current albumid, title and images. We wanted to do so because these values can change over time. However, as discussed in Section 2.0, the

number of requests necessary to do so would allow a single user to exhaust our pool of available requests. We discuss potential solutions in a later section.

[5.0] Flowchart: User Experience

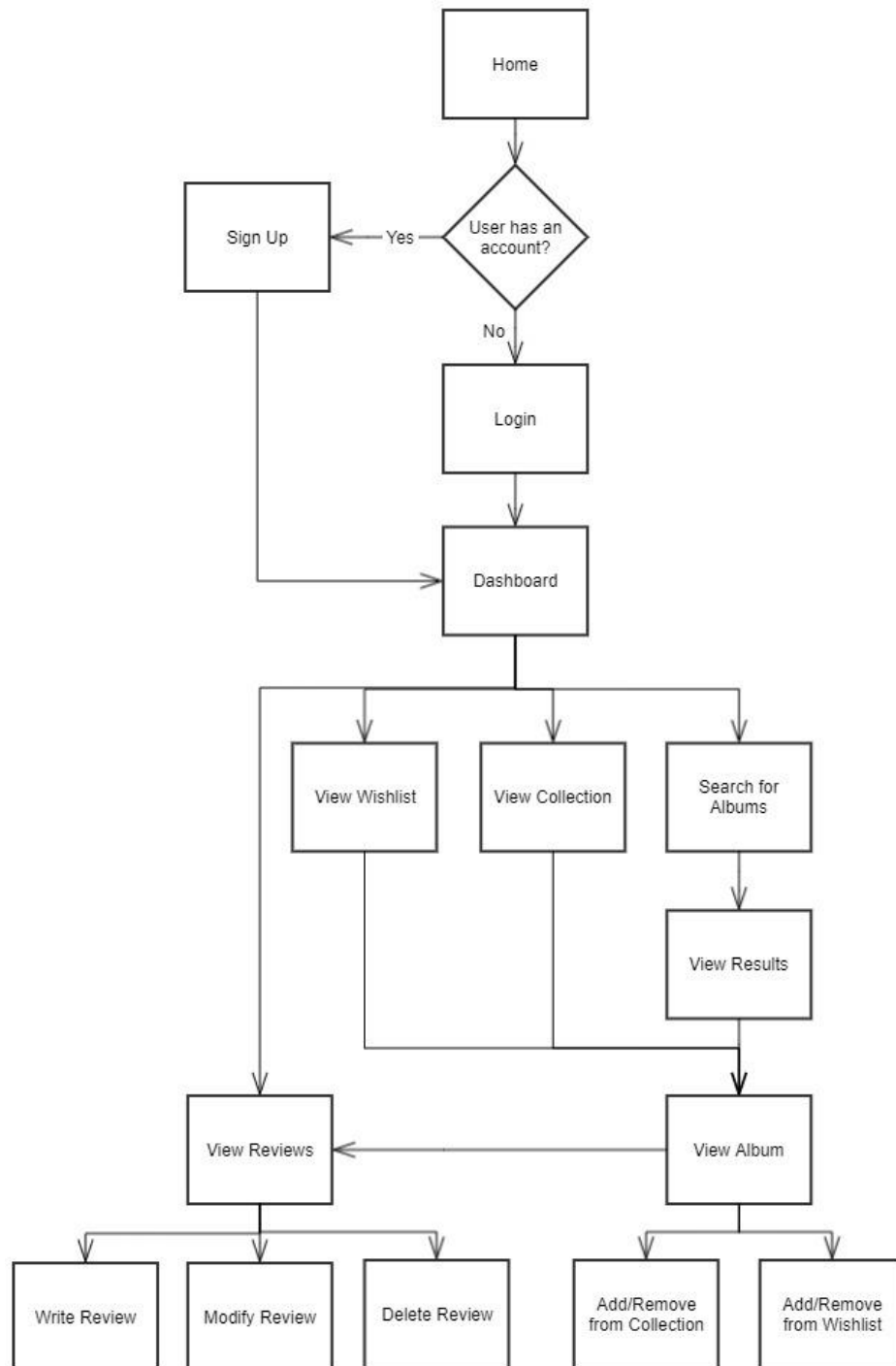


Figure 4.1 - Website Flowchart

The above diagram illustrates the path a user would take after arriving at the landing page.

After creating an account or logging in, the user is authenticated. All pages including and beyond the dashboard require a user to be authenticated. If not, they are redirected to the landing page.

At the dashboard, the user can choose to view their wishlist, collection and reviews. They can also search for albums. Selecting the wishlist or collection tab will display associated albums with their respective title and album art. Clicking on any of these albums will bring the user to the album page. Selecting the review tab will display a row consisting of the album art, title, and the user's review. Clicking on the album art will also allow the user to proceed to its respective album page.

When searching for albums, the user selects the criteria on which to search (barcode, catalogue number or title). Executing a search request will bring the user to the results page.

The results page displays a 5x10 grid of entries consisting of album art, album title, artist, and identifier (such as barcode or catalogue number). The user can execute a new search or click on an album to proceed to its respective album page.

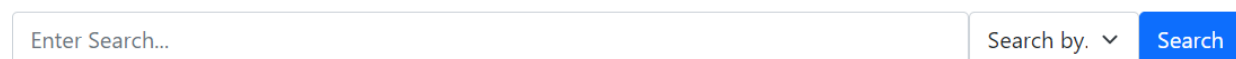
The album page displays information retrieved from the Discogs API that allow the user to identify their album (such as identifier, tracklist, etc). It also contains the lowest current Discog marketplace price for that album to allow the user to estimate its current worth. From this page, the user can add/remove the album from their wishlist/collection. They can also select the option to view reviews, which would allow them to proceed to the album's review page.

At the review page, the user can view all reviews of the album made by other users. If the album is currently in their collection, they have the option to write a review. Otherwise, they will be informed that they can only review albums currently in their collection. If an album is in their collection and they have not yet written a review, they are presented the option to write a review. If they have written a review, they are presented with the option to modify or delete a review. When selecting the option to modify a review, they are presented with a textarea form element that is pre-filled with their current review. Deleting a review removes the record from the database and presents them with the option to write a new review.

[6.0] Rubric Requirements

[6.1] Project allows user interaction with at least three different types of form elements (text box, select, radio, checkbox, etc)

Users interact with 3 types of form elements: textarea, input and select.



The image shows a search bar with a light blue border. On the left is a text input field with the placeholder text "Enter Search...". To the right of the input field is a dropdown menu labeled "Search by." with a downward arrow. Further right is a blue button with the text "Search" in white.

Among other methods, users will interact with input and select when searching for an album.

Your review:

Modify your review:

Hello

Submit

Users will interact with textarea when writing or modifying a review.

[6.2] Project uses at least three database tables with at least 10 fields (combined)

Our current implementation uses a database containing 5 tables and 19 combined fields.

[6.3] Project uses Web Storage or Sessions

This project uses express sessions.

```
const app = express();
app.set("view engine", "ejs");
app.use(express.static("public"));
app.use(express.urlencoded({ extended: true }));
app.use(
  session({
    secret: process.env.SES_SECRET,
    resave: true,
    saveUninitialized: true,
  })
);
```

The session is established after the user authenticates and is destroyed when the user logs out. Node will save information such as userid and authentication to session variables and retrieve them when necessary.

[6.4] Project allow users to update existing records in the database, in a friendly way (data is pre-filled)

When modifying a review, the textarea is prefilled with the value of the current review.

Your review:

Hello

Written by **You**

Modify

Delete

Your review:

Modify your review:

Hello

Submit

Submitting a modified review issues a call to a local API, which updates the existing record with the value of the new review.

[6.5] Project allow users to add records to the database

Users add records to the database in the following ways:

- Adding albums to their collection.
- Adding albums to their wishlist.
- Submitting a new review.
- Creating a user account.

[6.6] Project must have at least 100 lines of JavaScript/jQuery code

Not including comments, the project contains:

- 460 lines of code for Node
- 460 lines of code for client-side JavaScript & jQuery

[6.7] Project includes at least three local or external Web APIs. As part of your submission, please explain where the Fetch calls are.

We used the below APIs.

External:

- Discogs API: <https://www.discogs.com/developers>
 - /releases/{release_id}{?curr_abbr}
 - /database/seach?q={query}&{options}

Local Routes:

- /api/album/add
 - Used to add entries to the album table
- /api/collection/*
 - Used to get and modify records from the collection table
- /api/wishlist/*
 - Used to get and modify records from the update table
- /api/review/*
 - Used to get and modify records from the review table

What are fetch calls?

We are using the Mozilla MDN Web Docs to supplement our understanding of fetch calls. The `fetch()` method, provided by the Fetch API, allows us to make HTTP requests for resources, asynchronously. The `fetch` method takes a path to a resource, issues a Request to the target endpoint, and returns a Promise. Once the Response is received from the server, the Promise is resolved, providing us with either the requested resource or an error message.

[6.8] Project has a nice, professional, and consistent design, free of typos. Uses at least 50 CSS properties or Bootstrap.

The project uses around 61 properties for CSS as well as Bootstrap. We attempted to give it a professional look and feel.

[7.0] Open Issues & Known Bugs

We have recorded some of the issues or bugs that we have encountered but have not resolved during the time we had budgeted for implementation.

There is a delay in displaying the corresponding buttons when added or removing albums to a collection or wishlist.

- This occurs because the client will make a fetch request to an associated local API endpoint once the option is selected. We implemented this to check if the album is present in either table before issuing a query to the database.
- The idea is that a user can have many tabs of the same album open at once and submit several consecutive requests to either add or remove an album to either table.
- Our database is hosted on Heroku, so the delay occurs because (1) there are a maximum number of 10 concurrent connections and (2) the endpoint is slow to respond to requests.

All available options are displayed on the review page while the page loads. Correct options are displayed after the page is loaded.

- This occurs because the buttons are initialized as showing as the page loads. They are hidden by jQuery commands that are issued after the document is finished loading.
- A simple fix would be to create a `“.hidden”` CSS class and add it to all the buttons on the `review.ejs` file. After the page is loaded, we would instead use jQuery to show the valid options.
- We did not immediately notice this issue because our local development environment is much faster than the hosted one. However, we chose not to resolve this issue due to time constraints.

Album entries are only added and not removed.

- We added the album table to reduce the number of fetch requests made to the Discogs API. However, we are still determining how long to cache each record and the mechanism for updating them.
- Over time, the records may become inaccurate. The table could eventually approach the Discogs' database in size as entries are added. In a worst-case scenario, the table would increase beyond the size of the Discogs' database they change the albums associated with their albumids, for example.

- A potential solution would be to update the records in the album table on a regular basis, distributing fetch requests so that service remains available to our users. Records with albumids that are not present in any collection or wishlist can be purged. Otherwise, title and resource URL are updated if the content of the fetch request differs from the content of the table during scheduled updates. We are not yet sure what to do if Discogs changes their albumids.

[8.0] Future Implementation (Conclusion)

Due to the time limitations of our project (we designed and implemented this website in one week), there are unaddressed items that we would like to pursue after submission.

First, we need to identify a method for updating and purging records from the album table for the reasons we discussed in Section 7.0.

Second, we want to re-evaluate our understanding of the Discogs API to find a method to display an accurate representation of the total value of the user's album collection.

Third, we want to integrate OAuth into the website's authentication system. This would allow a user to browse the Discogs marketplace, make sales and purchase albums.

Fourth, we want to fully address any other open issue that we are aware of but were not able to address within the project's timeframe.

Last, we want to continue to improve the website's functionality and styling to be mobile responsive.

[9.0] Screenshots

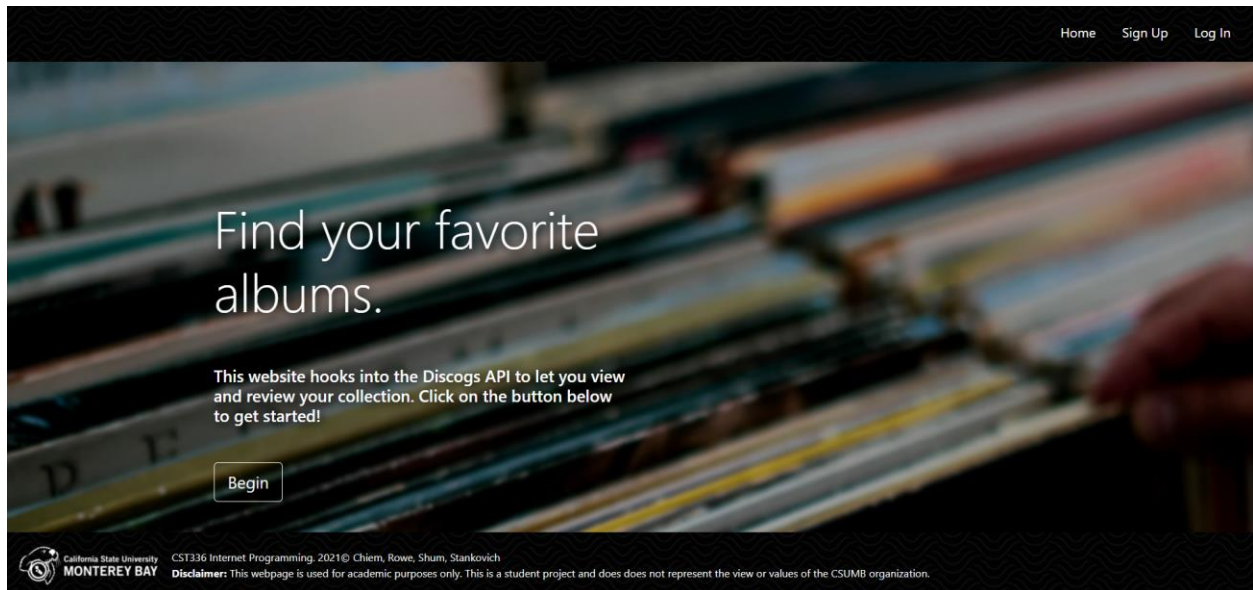


Figure 9.1 - Landing Page

[Home](#) [Sign Up](#) [Log In](#)

Sign up, today!

First Name:

Last Name:

Email:


Username:


Password:

Confirm Password:

☐ I'm not a robot re

[Sign Up](#)





CST336 Internet Programming, 2021 © Chien, Rowe, Shum, Stankovich
Disclaimer: This webpage is used for academic purposes only. This is a student project and does not represent the view or values of the CSUMB organization.

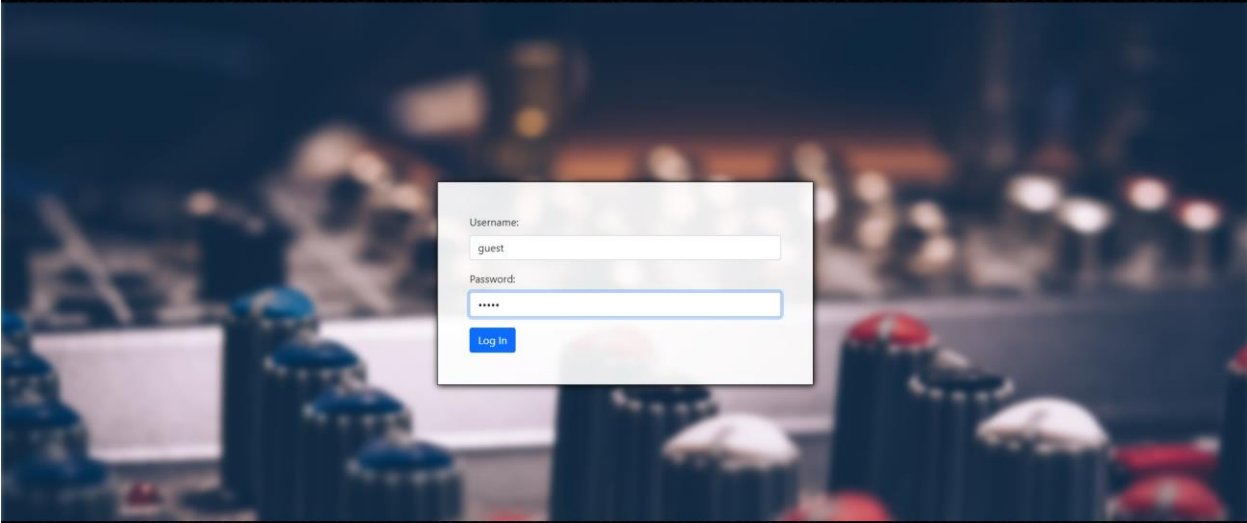
Figure 9.2 - Sign Up


[Home](#) [Sign Up](#) [Log In](#)

Username:

Password:

[Log In](#)





CST336 Internet Programming, 2021 © Chien, Rowe, Shum, Stankovich
Disclaimer: This webpage is used for academic purposes only. This is a student project and does not represent the view or values of the CSUMB organization.

Figure 9.3 - Log In

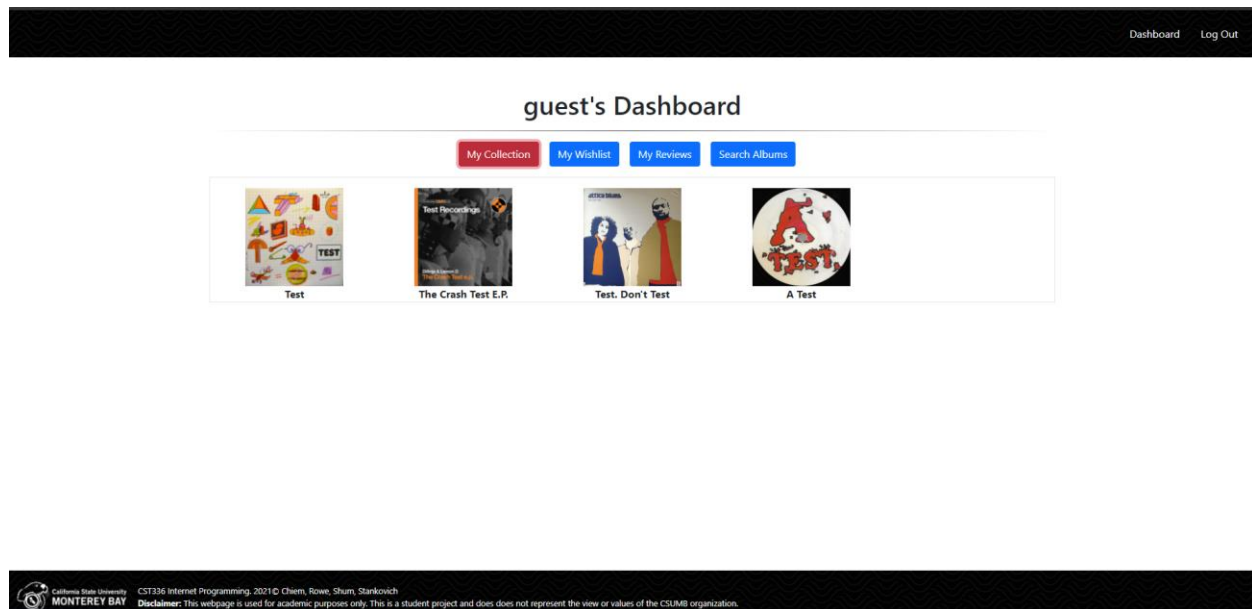


Figure 9.4 - Dashboard – Collection

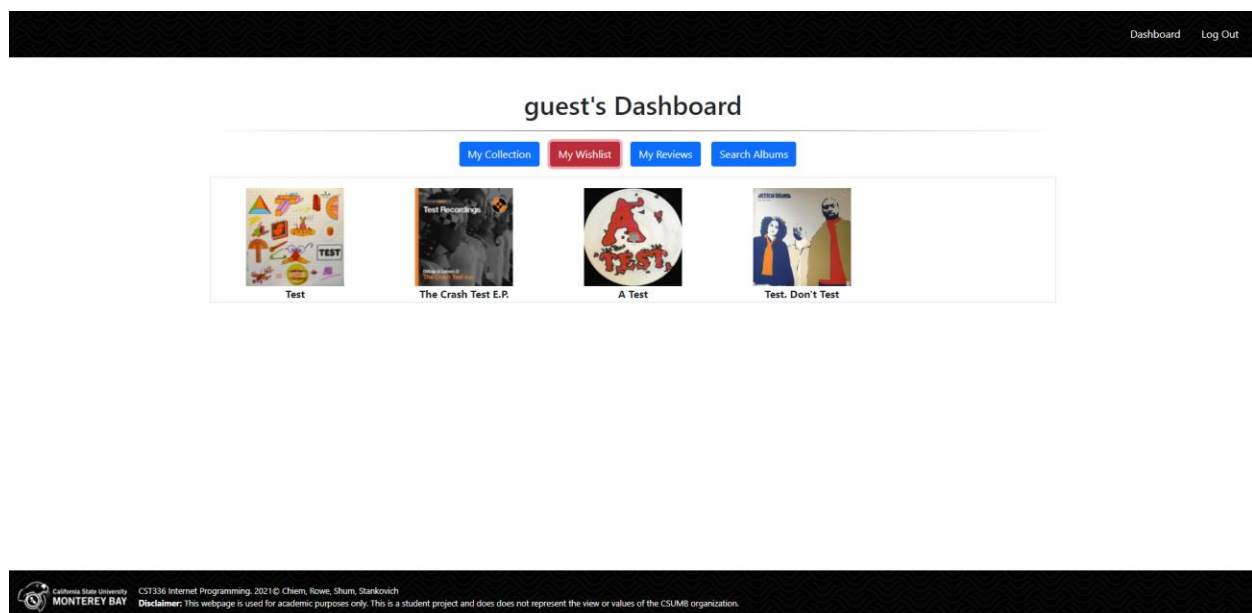


Figure 9.5 - Dashboard - Wishlist

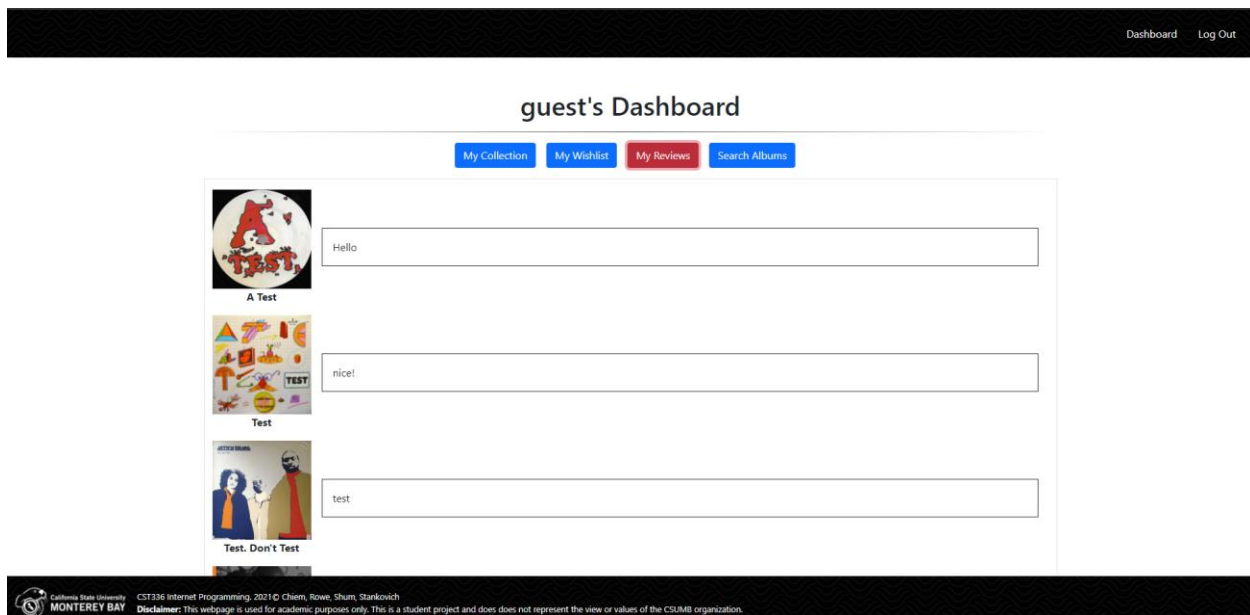


Figure 9.6 - Dashboard – Review

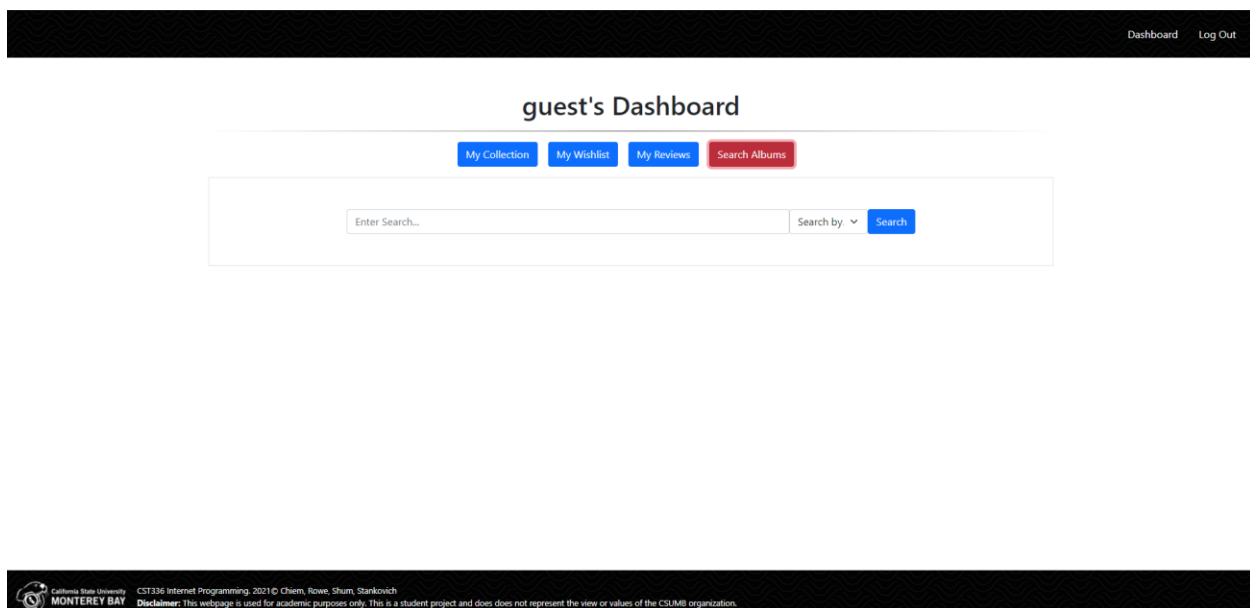


Figure 9.7 - Dashboard - Search

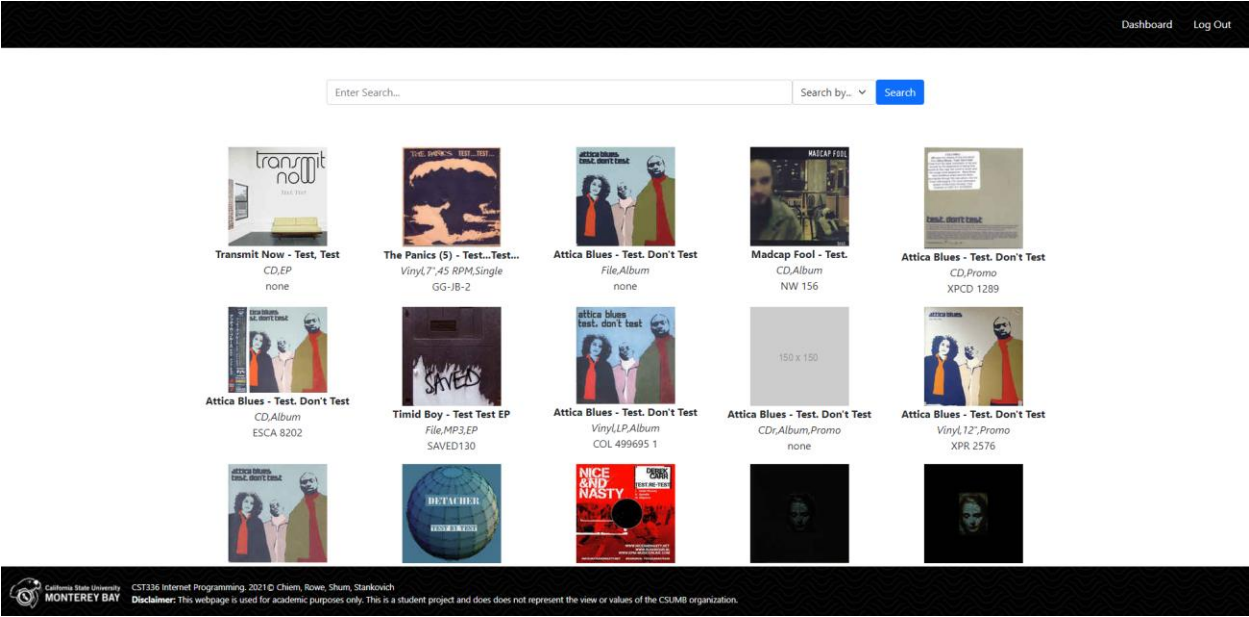


Figure 9.8 – Search Results

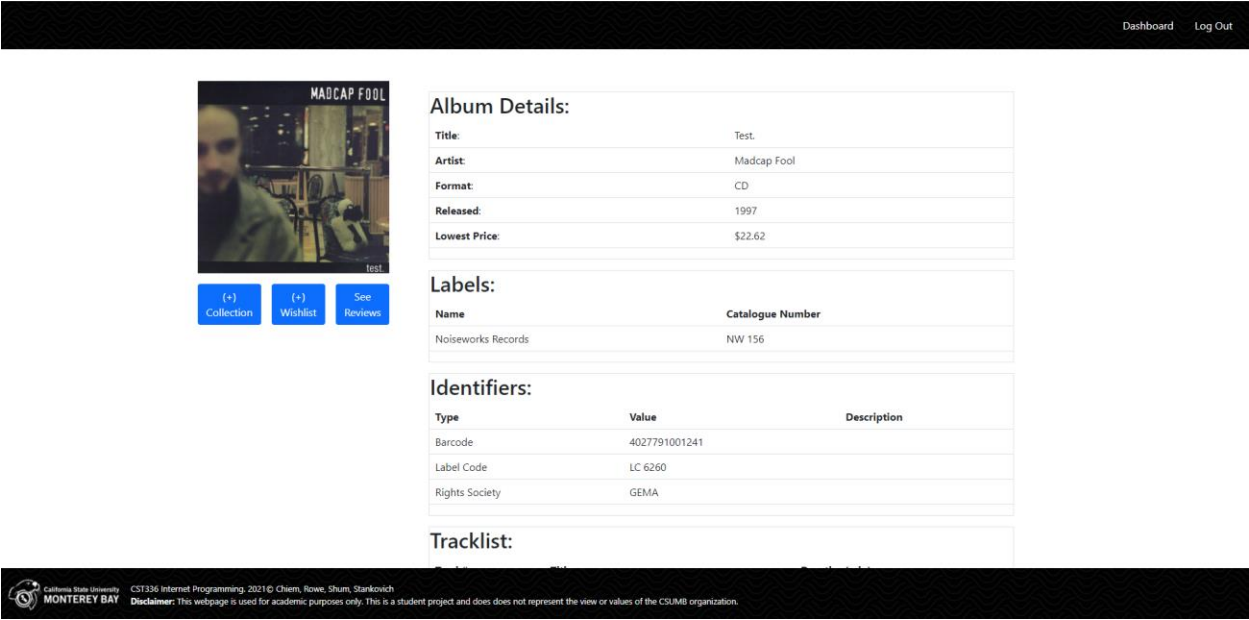


Figure 9.9 - Album Page

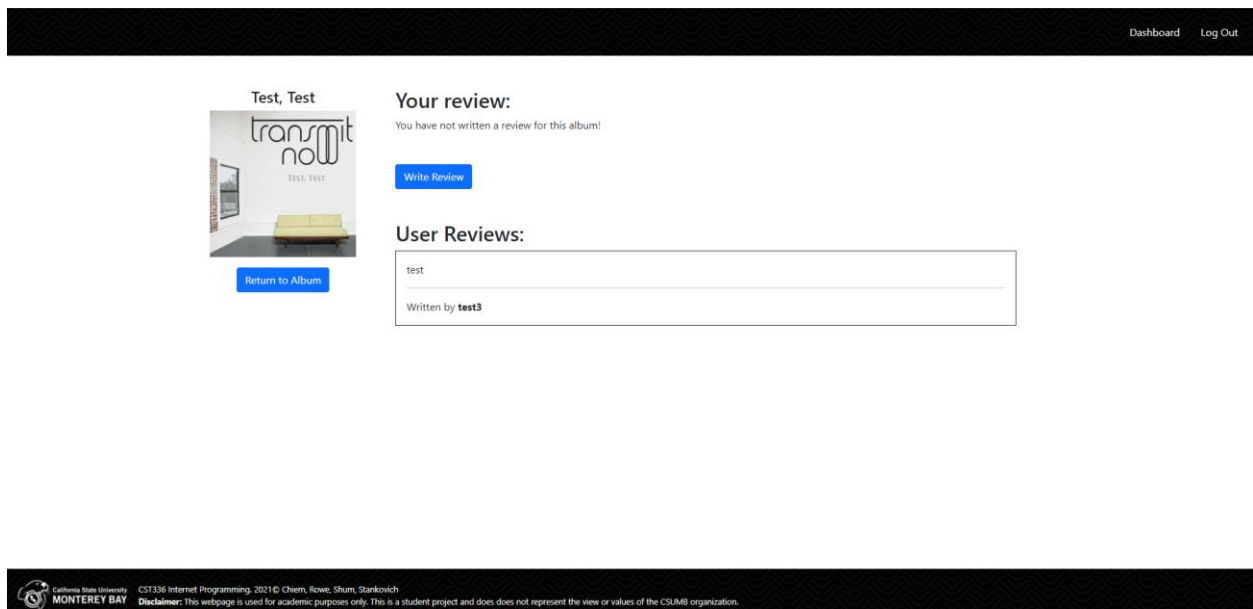


Figure 9.10 - Review Page (None Written)

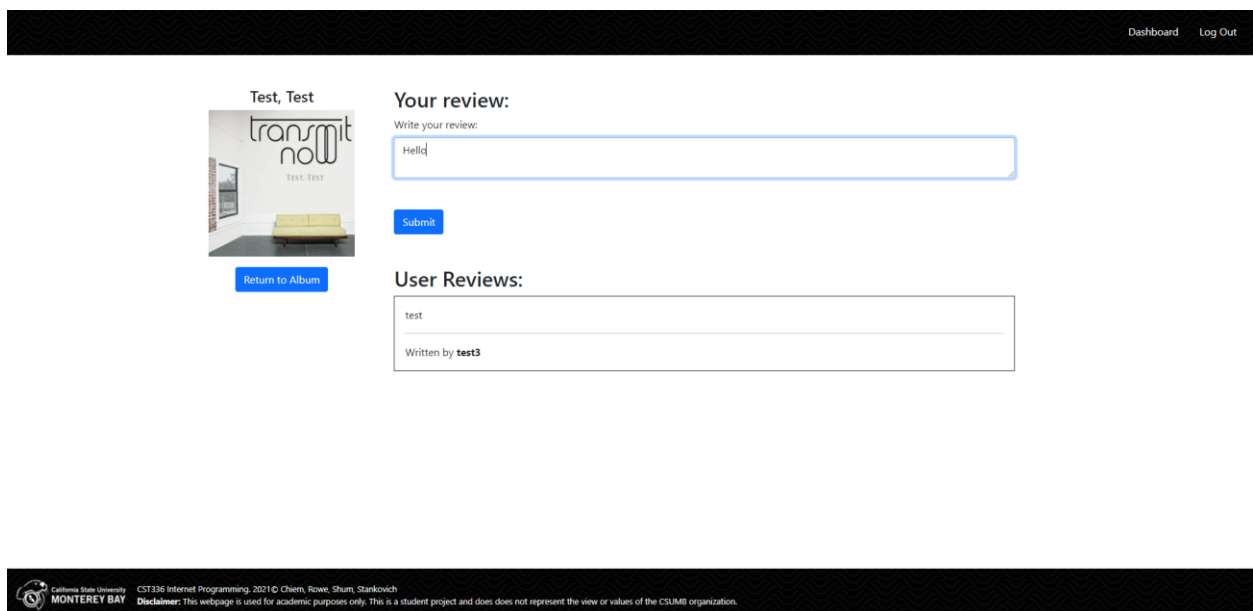


Figure 9.11 - Review Page (Write)

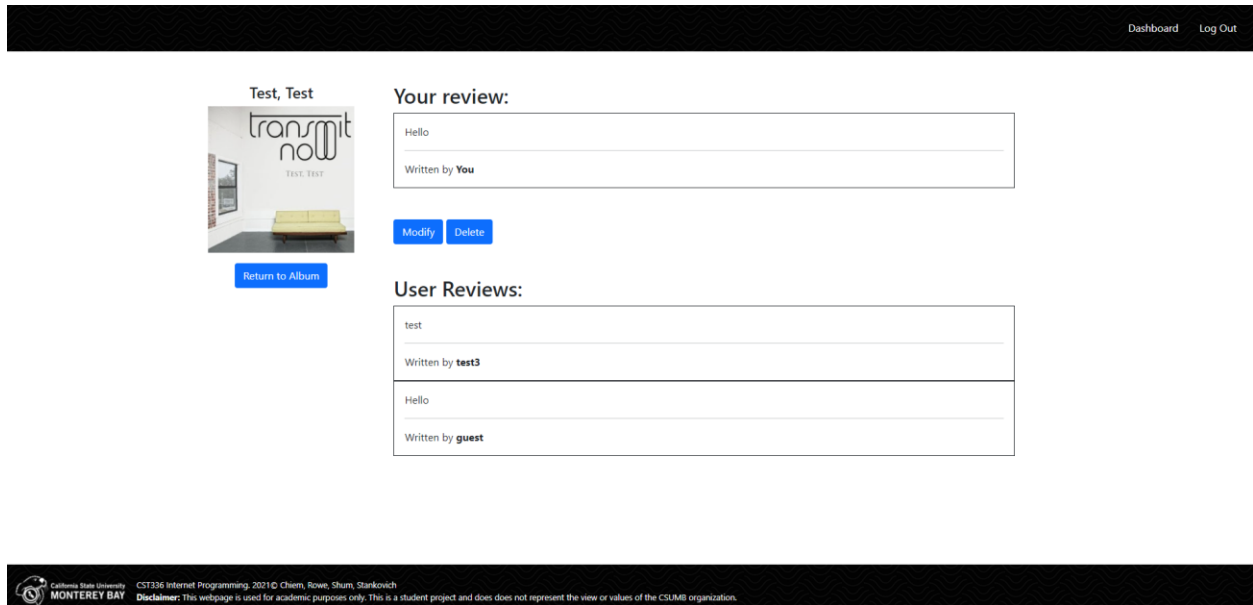


Figure 9.12 - Review (Already Written)

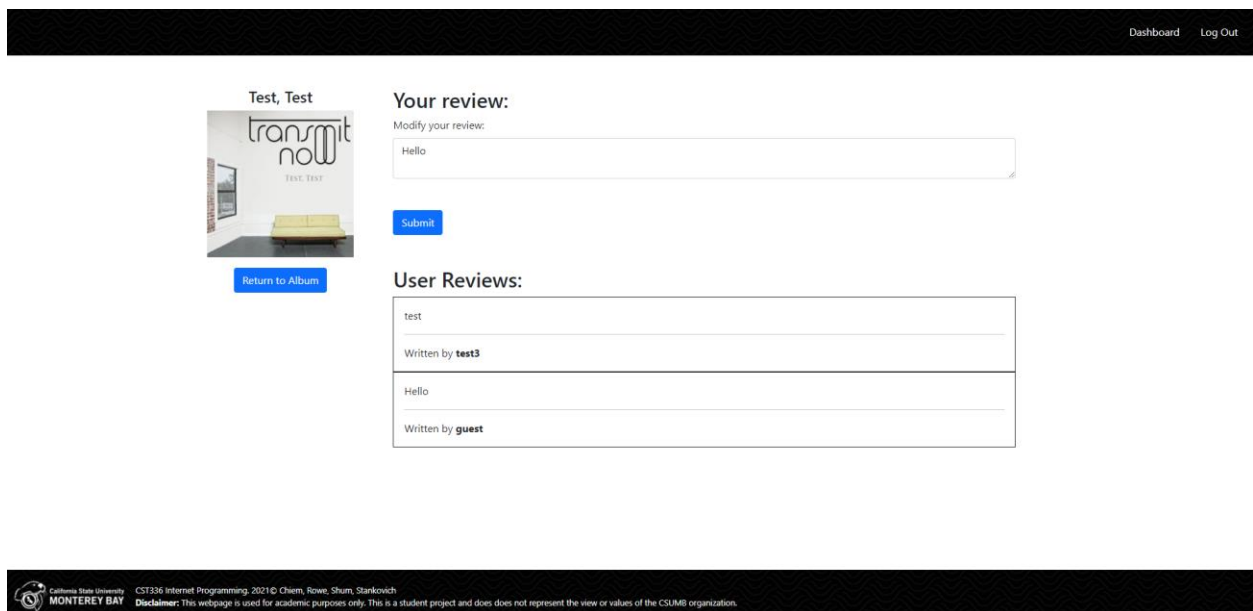


Figure 9.13 - Review (Modify)