**Personal Playlist Share & Rate**

**Development Book**

ISTE 432

Daniel Chung

Hannah Mallari

Matt Marchinetti

Sachi Nutulapati

Kajal Nagrani

**Table of Contents**

[**Team Members**](#_u9i259stoeso) **3**

[**Background**](#_cslkcn51bbsw) **3**

[**Project Description**](#_n0kr1sz2ray9) **4**

[**Project Requirements**](#_67u2ouazqfr2) **4**

[**Business Rules**](#_y8zj0neqw3ht) **4**

[**Technologies Used**](#_5l01q3hszcvv) **5**

[**Timeline**](#_p82ap981bhze) **5**

[**Design Patterns**](#_pl3kknx55aft) **6**

[**Layering**](#_e8z8sgdd49ep) **10**

[**Changes**](#_mkc2rc8d9ar5) **13**

## Team Members

|  |  |
| --- | --- |
| Team Members | Role |
| Daniel Chung | Jasmine #1 |
| Hannah Mallari | Jasmine |
| Matt Marchinetti | Mulan 3000 |
| Sachi Nutulapati | Mulan 1.0 |
| Kajal Nagrani | Ariel |

## Background

The client for this project is a radio host and “social media personality.” Her radio show listeners and other fans are mainly interested in her music taste and want to have access to her song choices outside of just her show. To cater to this, she has been sharing her Spotify playlists on her social media. She has seen an opportunity to create something that she knows people want - a platform for sharing playlists. While it is possible to follow users on Spotify and see their playlists, you cannot rate or comment on the quality of these playlists, and it is hard to look for playlists created by users unless you are looking for something particularly specific. The interest for these features can also be seen in articles like “Barack Obama’s Best-Of 2019 Playlist Boasts Beyonce, Bruce Springsteen, Rosalia” put out by Rolling Stone and the popularity of sites like SoundCloud or old sites like 8tracks and Myspace.

## Project Description

The goal of this project is to use the Spotify API to extend the social capabilities of Spotify, specifically the playlist and song sharing. The only way to share playlists with other people currently is to share them on social media or if you find public ones on Spotify based on a genre/related song that you are browsing. The idea here is to be able to share your playlists on your page and other users can listen to the playlist on the Spotify application, rate the playlist, and leave comments.

## Project Requirements

The client would like a site with the following functionality: log-in, commenting, rating, modifying, tagging, importing and exporting playlists. Playlists should be exported to and imported from Spotify. Playlists imported on the site can be viewed, rated, and commented on by anyone else with an account. A user must have a Spotify account to log-in to this site, so user log-in needs to be authenticated through the Spotify API. A user has the ability to delete and add songs to their own playlists, as well as delete whole playlists from their profile, and delete comments made by others on their (the user’s) playlists.

## Business Rules

* A Spotify account is required
* Non-Spotify users have no privileges
* Spotify users can view, modify, comment, rate, import, and export playlists
* Rating is out of 5 stars
  + Lowest rating allowed is 1 out of 5
  + Highest rating allowed is 5 out of 5
* Playlists should be created by a user for them to be imported/exported
* Comments are 140 characters maximum

## 

## Technologies Used

* Node JS
* Express
* React
* Spotify API
* HTML/CSS
* MySQL

## 

## Timeline

|  |  |
| --- | --- |
| **Date** | **To Do** |
| 02/21 | (Milestone 2) Design |
| 02/28 | (Milestone 3) Layering |
| 03/30 | (Milestone 4) Exception Handling |
| 04/03 | (Milestone 5) Refactoring |
| 04/17 | (Milestone 5) Testing |
| 04/24 | (Milestone 6) Packaging |
| 04/27 | Project finalized |

## 

## Design Patterns

Proxy Pattern

Proxy patterns allow systems to interact with a secondary system or service without having to contain functionality for contacting that secondary system. They also allow for the opportunity to add heightened security, logging or networking. Many of our data layer objects will be interacting with the Spotify API without the rest of the system understanding how the underlying functionality is operating. This allows the rest of the system to utilize information stored remotely without containing logic for reaching out to a secondary system.

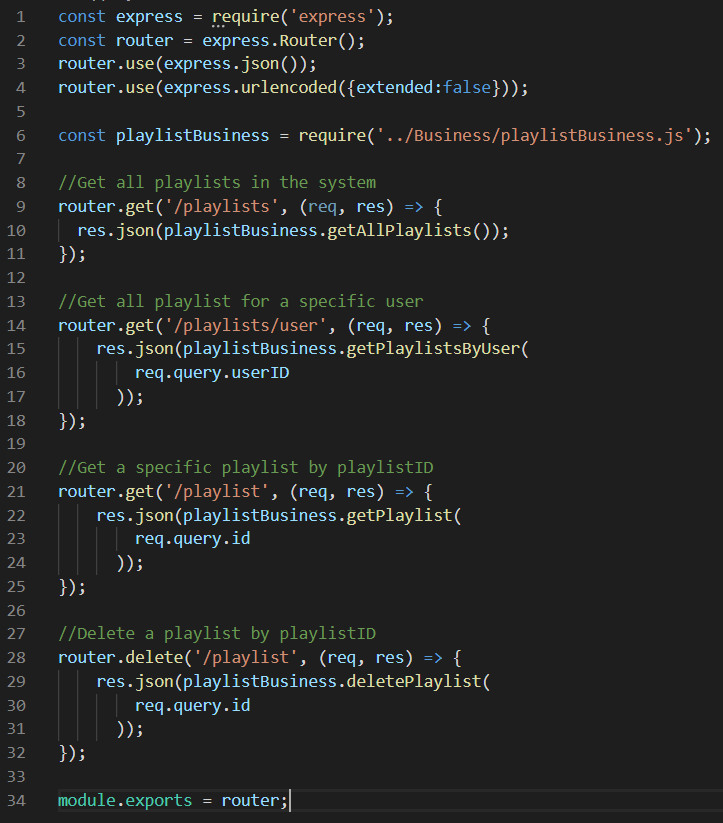
Singleton Pattern

The Singleton Pattern ensures that there is only ever one instance of a class, it also provides a way to access this class globally throughout the code. This class allows us to prevent any other class from creating another instance of the class. This could be used to access common authentications throughout our code. For example, making sure that all comments in a playlist are less than 140 words. We can call a function that checks that from anywhere in the code.

(Edited 03/29/2020. See: Changes section)

Facade Pattern

A facade pattern provides an abstract, higher level grouping of functionalities which conceals the complexity of the system it sits on top of. Most API’s conform to effectively creating a facade pattern via the service layer by exposing high level functionalities such as getPlaylist or updateUser. While these are exposed as fairly straight-forward for a separate application to utilize, there are many steps in the business layer which confirm such things as if the user has the authority to do so, if the passed-in parameters are valid, if it conforms to business rules, etc.

Code Snippet:

This code snippet is a simplified version of what we expect one of our service layer objects to look like on the API we are building. API’s inherently resemble the facade pattern because the business and data layers use a combination of functionalities to validate, authorize, and retrieve or modify data but the service layer organizes all of that into straight-forward endpoints for external systems to utilize.

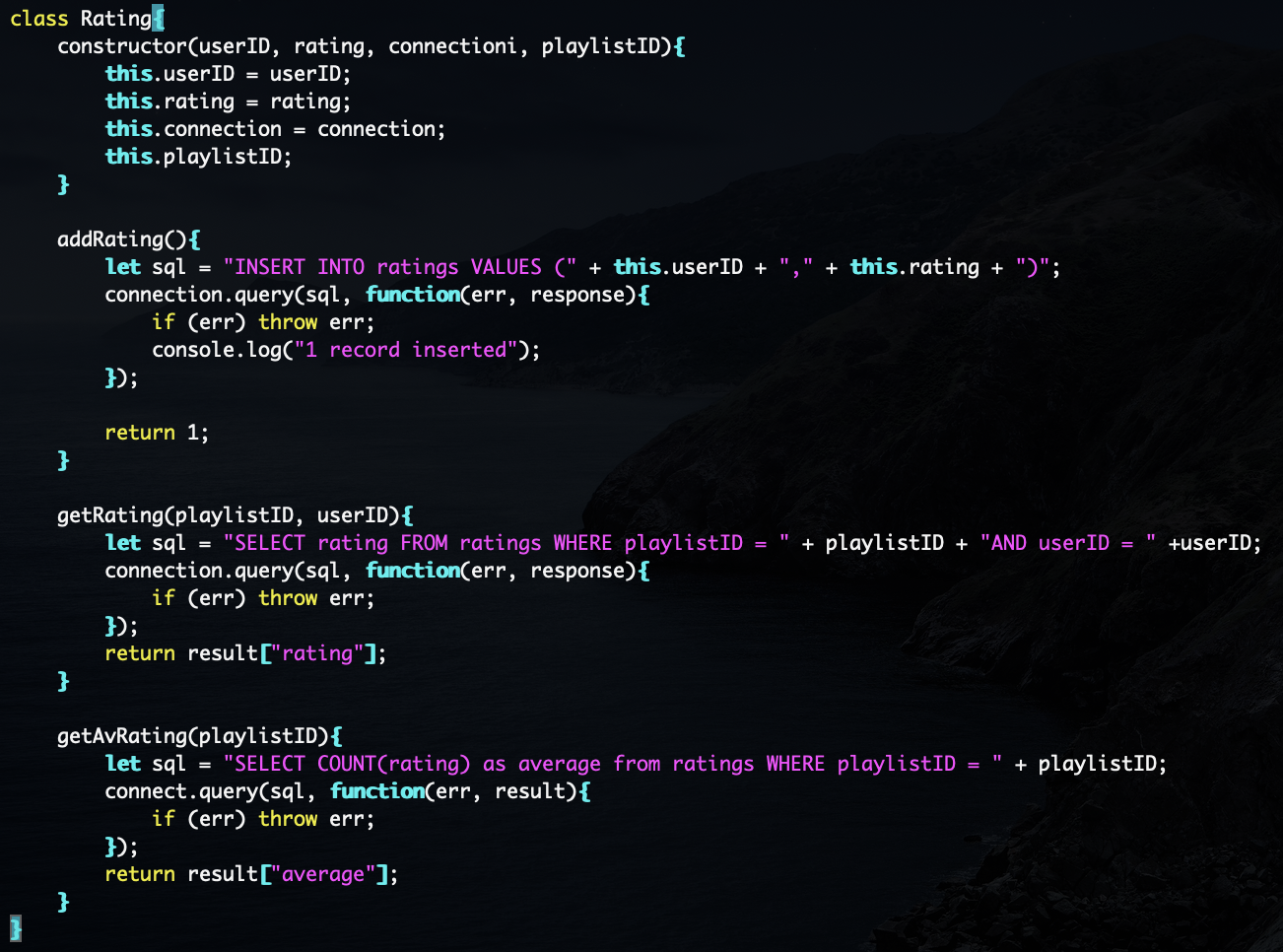
Command Pattern

The command pattern is a way for an invoker to store a concrete implementation of an abstract command object, which allows for easier execution of a command such as executing a request to an API, storing the response object and implementing a way to parse the response to retrieve the data necessary to display to the user. The command contains the information from the invoker such as the URL to hit and the parameters to hit with. The execute function on the abstract command would concatenate the parameters and the URL to send and store the response.

## Layering

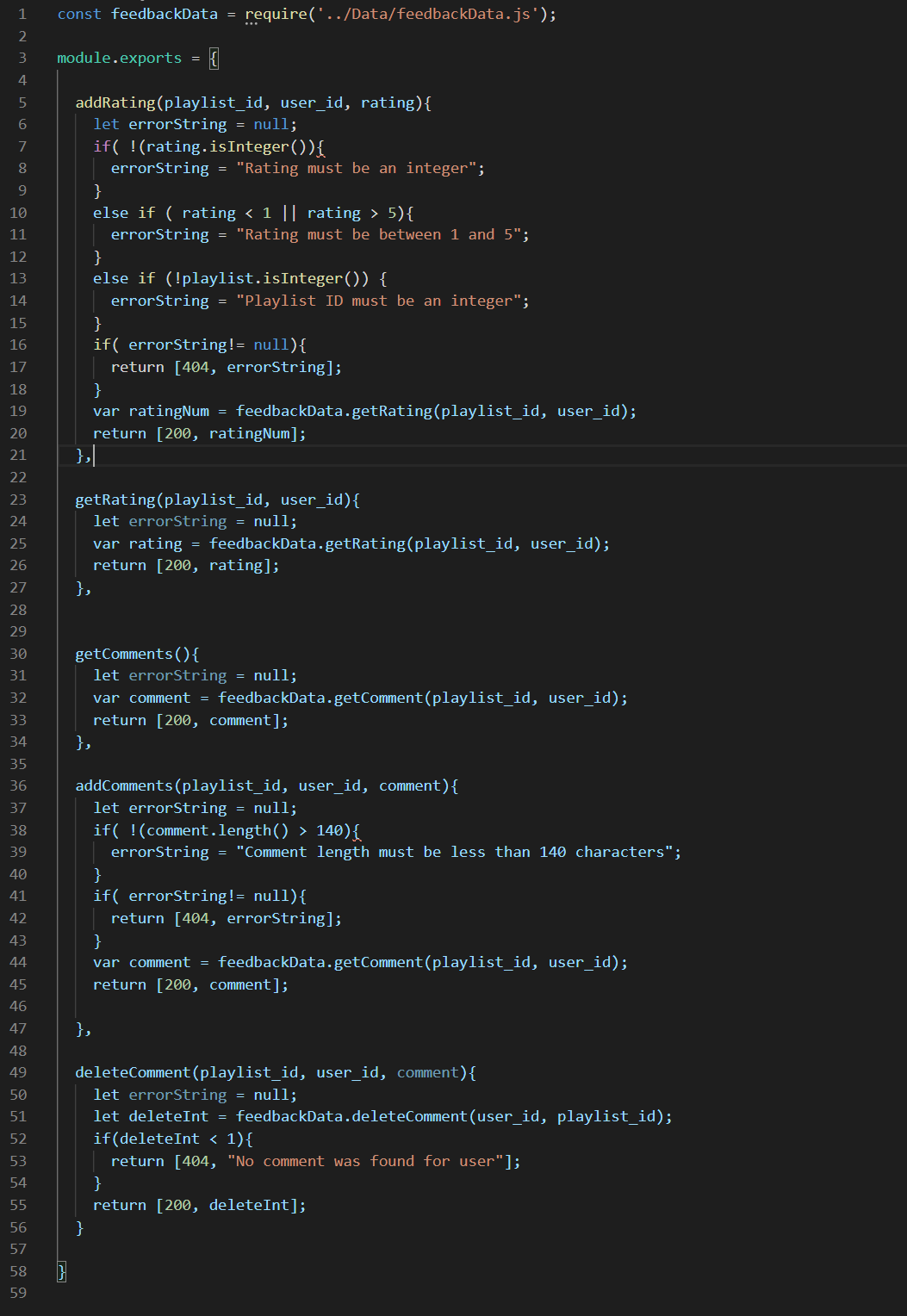
Data Layer

The data layer will contain separate classes for each subsystem involved in our project including UserData, PlaylistData, CommentData and RatingData classes as well as a soft requirement for a TaggingData class to provide heightened usability for our users. The peculiarity here is that UserData and PlaylistData will primarily be interacting with the Spotify API in a proxy pattern whereas the CommentData, RatingData and TaggingData classes will be utilizing a combination of both Spotify API functionality and information from our local database.



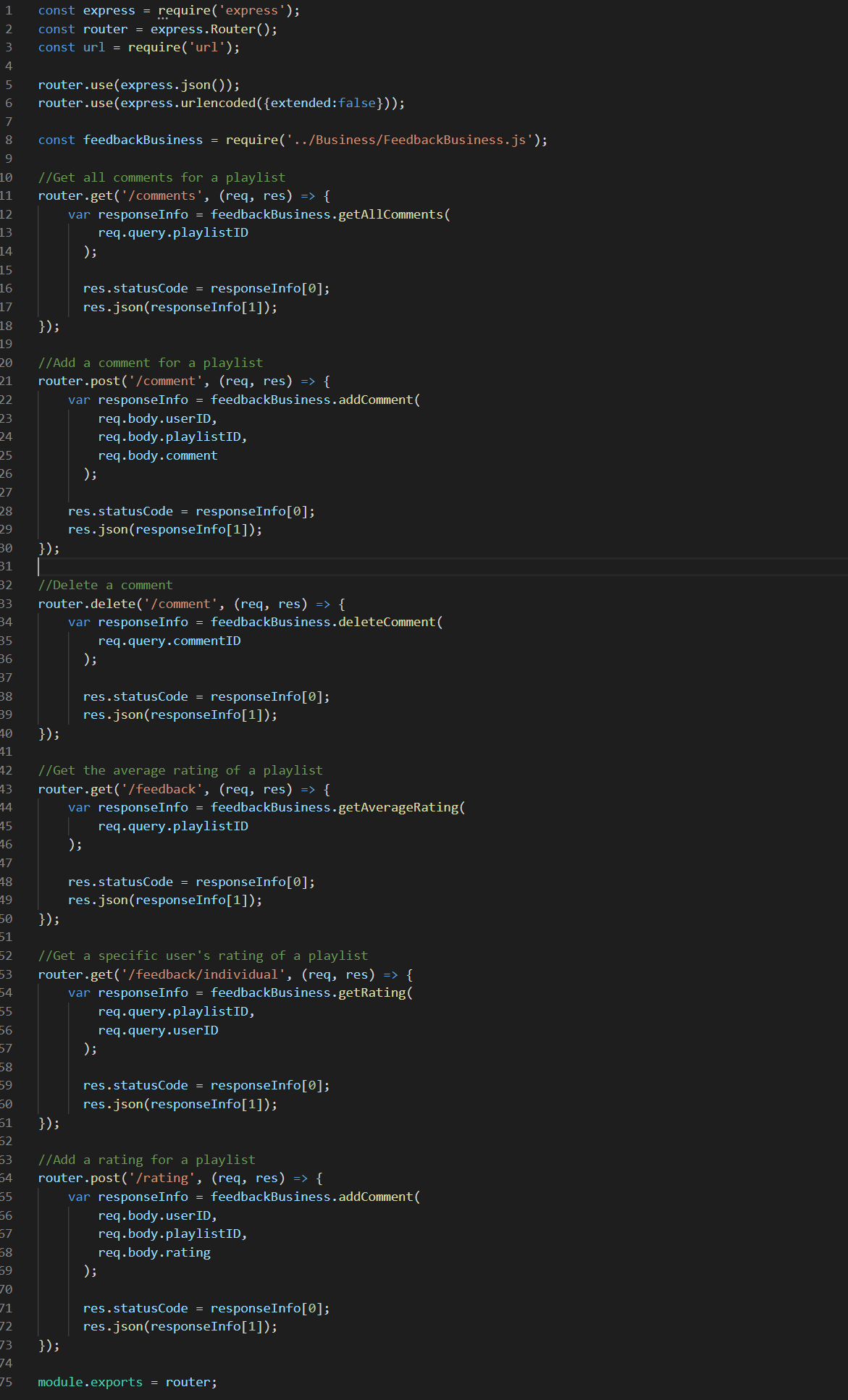
Business Layer

The business layer will have separate classes for all of the subsystems in our project which will be UserBusiness, PlaylistBusiness, and FeedbackBusiness. Our soft requirement will be TaggingBusiness class which will improve the usability for users. These classes will enforce the business rules. The UserBusiness class will enforce that you have to have a Spotify account and be logged in. The PlaylistBusiness class will import, create, get, update and search playlists. The FeedbackBusiness class will add, delete comments. The class will also add and update ratings.



Service Layer

The service layer will contain separate classes including the UserService, PlaylistService, and FeedbackService. The playlist and feedback will include CRUD functionality. In the playlist class, a user can import, create, get, and update a playlist. A user can get all the playlists from one user and also add and delete songs. Other functionality includes searching for a playlist by name and description or searching by tags. The UserService class includes a notification activity for followers as well as getUser and login methods.



Presentation Layer

The user will have access to X amount of views including: a log-in page, their dashboard featuring the users that follow, a list of their own playlists, a playlist view that shows them the playlist’s songs, name, description, comments, and average rating, and an edit page for a given user’s playlist.

## Exceptions

In order to ensure that the users never handle any exceptions, it was decided that all exceptions are to be handled in the business and data layers, typically in the business layer. Most of what we plan to be dealing with are HTTP responses, where-in they will be caught by the backend and then passed to the front end to be dealt with.

400 (Bad Request)

* Invalid JSON format
  + Invalid syntax exception

401 (Unauthorized)

* Incorrect credentials
  + Caught by backend -> Passed to front end, front end has to reroute to log in page and say invalid credentials

403 (Forbidden)

* Logged-in, can’t do that
* Trying to modify a comment on a playlist that isn’t yours
* Trying to modify a rating on a playlist that isn’t yours
* Trying to modify a playlist that isn’t yours

404 (Not Found)

* Playlist is not found because it has been deleted or removed from user’s account
* User is not found because they deactivated their account
* Rating is not found because it was deleted by the user
* Comments are not found because they have been blocked or deleted by the user or owner of the playlist

405 (Method not Allowed)

* If something is deleted that user didn’t intend/didn’t know about

409 (Conflict)

* If you have already made a rating & comment on a playlist (multiple ratings from one user on a given playlist not allowed)

500 (Internal Server Error)

* SQL Exceptions : Database not found

## Changes

02/28/2019 - Added Proxy Pattern to the design patterns section.

03/29/2020 - Got rid of the goal of having child comments/being able to respond to comments. Only going to allow one comment and one rating from a user.

> Deleted from Design Patterns section:

Composite Pattern

The composite pattern treats multiple objects as a single instance of the same object. This is useful for our application in the comments section. Users can comment on existing comments so the entire comment thread is considered as one instance instead of separate comment objects. Right now, we are unsure whether or not we should divide up the comments by parentComment and childComment which would require us to implement the factory pattern or keep the thread as one instance.

Code Snippet:

This snippet is a sample class that displays that our comments will be resembling a composite pattern. The comment class will store a list of nested comments which can be identified as either a composite or leaf by testing if the childComments list is empty.

Factory Pattern

The factory pattern allows for easier construction of similar but separate objects often by providing overloaded methods or providing methods that can intelligently decide which type to construct. This may be required for our system to support the creation of nested comments by constructing either a parentComment or childComment according to the information present.