Lab 6

Start Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip

CS-546 Lab 6

A Movie API

For this lab, you will create a simple server that provides an API for someone to Create, Read, Update, and Delete movies and also movie reviews.

We will be practicing:

- Seperating concerns into different modules:
- Database connection in one module
- Collections defined in another
- Data manipulation in another
- Practicing the usage of **async / await** for asynchronous code
- Continuing our exercises of linking these modules together as needed
- Developing a simple (9 route) API server

Packages you will use:

You will use the **mongodb** (https://mongodb.github.io/node-mongodb-native/) package to hook into MongoDB

You may use the **lecture 4 code** (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_04/code) and the **lecture 5 code** (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_05/code) and **lecture 6 code** (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_06/code) as a guide.

You can read up on **express** (http://expressjs.com/) on its home page. Specifically, you may find the **API Guide section on requests** (http://expressjs.com/en/4x/api.html#req) useful.

You must save all dependencies you use to your package.json file

Folder Structure

YOU MUST use the directory and file structure in the code stub or points will be deducted. You can download the starter template here: **Lab6_stub.zip** (https://sit.instructure.com/courses/61549/files/10389144/download?download_frd=1)

PLEASE NOTE: THE STUB DOES NOT INCLUDE THE PACKAGE.JSON FILE. YOU WILL NEED TO CREATE IT! DO NOT ADD ANY OTHER FILE OR FOLDER APART FROM PACKAGE.JSON FILE.

Database Structure

You will use a database with the following structure:

- The database will be called **FirstName_LastName_lab6**
- The collection you use to store movies will be called `movies` you will store a sub-document of `reviews`

movies

The schema for a movie is as followed:

```
{
    _id: ObjectId,
    title: string,
    plot: string,
    genres: [strings],
    rating: string,
    studio: string,
    director: string,
    castMembers: [strings],
    dateReleased: string representation of a date in mm/dd/yyyy format
    runtime: string,
    reviews:[], (an array of review objects, you will initialize this field to be an empty array when a movie is created),
    overallRating: number (from 0 to 5 this will be a computed average from all the movie reviews posted for a movie,
    the initial value of this field will be 0 when a movie is created)

}
```

The `_id` field will be automatically generated by MongoDB when a movie is inserted, so you do not need to provide it when a movie is created.

An example of how Hackers would be stored in the DB:

```
{
    _id: ObjectId("507f1f77bcf86cd799439011"),
    title: "Hackers",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[],
    overallRating: 0
}
```

The Movie Review Sub-document (stored within the movies document)

```
{
  _id: ObjectId,
  reviewTitle: string,
  reviewDate: string (string value of a date in MM/DD/YYYY format),
  reviewerName: string,
  review: string,
  rating: number 1-5 (floats will be accepted as long as they are in the range 1.5 or 4.8 for example. We will only use one decimal place)
}
```

An example of the review sub-document:

```
{
  _id: ObjectId("603d992b919a503b9afb856e"),
  reviewTitle: "Meh...",
  reviewDate: "10/13/2022",
  reviewerName: "Patrick Hill",
  review: "This movie was good.  It was entertaining, but as someone who works in IT, it was not very realistic",
  rating: 3.5
}
```

data/movies.js

In movies, you will create and export 5 methods. Create, Read (one for getting all and also one getting by id), Update, and Delete. **You must do FULL error handling and input checking for ALL functions as you have in previous labs, checking if the input is supplied, correct type, range etc. and throwing errors when you encounter bad input. You can use the functions you used for lab 4 however, you will need to create an updateMovie function and also modify your createMovie function. renameMovie from lab 4 will not be used.**

As a reminder, you will keep the same function names you used in lab 4:

```
createMovie(title, plot, genres, rating, studio, director, castMembers, dateReleased, runtime)
getAllMovies()
getMovieById(id)
removeMovie(id)
updateMovie (id, title, plot, genres, rating, studio, director, castMembers, dateReleased, runtime)  Note: th
is is the new function you will create
```

For the functions you did in lab 4, all the same input requirements apply in this lab, for new update function, those requirements are stated below.

NOTE: `overallRating` and reviews are not passed into either createMovie or updateMovie functions.

For createMovie: When a movie is created, in your DB function, you will initialize the reviews array to be an empty array. You will also initialize overallRating to be 0 when a movie is created.

For updateMovie: you will not modify the overallRating or reviews in the updateMovie function, you must keep them intact as they were before the update.

async updateMovie(id, title, plot, genres, rating, studio, castMembers, dateReleased, runtime)

This function will update **all** the data of the movie currently in the database.

If `id, title, plot, genres, rating, studio, director, castMembers, dateReleased, runtime` are not provided at all, the method should throw. (**All fields need to have valid values**);

If `id, title, plot, rating, studio, director, dateReleased, runtime` are not `strings` or are empty strings, the method should throw.

If `id` is not a valid `ObjectId`, the method should throw.

If `genres` is not an array that has at least one string element contained in it, this method will throw.

if any of the elements in `genres` is not a valid string (empty strings or strings with just spaces are invalid), the method should throw.

If `castMembers` is not an array that has at least one string element contained in it, this method will throw.

if any of the elements in `castMembers` is not a valid string (empty strings or strings with just spaces are invalid), the method should throw.

If `dateReleased` is not a valid date string (09/31/2019 is not valid as there are not 31 days in September. 02/30/2020 is not valid as there are not 30 days in February this field MUST be a valid date), or if `dateReleased` is less than 01/01/1900 or greater than the current year + 2 (2024 in this case) the the method should throw. You do not have to take leap years into account and the format of the date must be in mm/dd/yyyy format.
Note: so only years 1900-2024 are valid values. Do not hardcode the year to be 2024, use the current year and then add 2 years to it. This will ensure that the application will function past 2024).

`runtime` MUST be in the following format "#h #min". both #'s must be a positive whole number, but for minutes, that may be zero but the max value for min should be 59 since 60min would be 1 hour:

For example: valid: "2h 30min", "2h 0min", "1h 59min" not valid: "-5h 20min", "3.5h 10min", "0h 30min" (most movies are longer than an 30 min, and usually longer than 1 hour), "2h 60min" (this should just be 3 hours). This field will be case sensitive so you MUST match the format shown exactly.

If the update succeeds, return the entire movie object as it is after it is updated.

data/reviews.js

In reviews, you will create and export 4 methods. Create, Read (one for getting all and also one getting by id), and Delete. Y**ou must do FULL error handling and input checking for ALL functions as you have in previous labs, checking if input is supplied, correct type, range etc. and throwing errors when you encounter bad input.**

Function Names:

`createReview(movieId, reviewTitle, reviewerName, review, rating) Note: Notice we are not passing reviewDate, this will be set to the current date when the review is created`
`getAllReviews(movieId)`
`getReview(reviewId)`
`removeReveiw(reviewId)`

async createReview(movieId, reviewTitle, reviewerName, review, rating);

This async function will return to the newly created review object, with **all** of the properties listed in the above schema.

If `movieId, reviewTitle, reviewerName, review, rating` are not provided at all, the method should throw. (**All fields need to have valid values**);

If `movieId, reviewTitle, reviewerName, review` are not `strings` or are empty strings, the method should throw.

If the `movieId` provided is not a valid `ObjectId`, the method should throw

If the movie doesn't exist with that `movieId`, the method should throw.

If `rating` is not a number from 1 to 5, the method should throw (floats will be accepted as long as they are in the range 1.5 or 4.8 for example. We will only use one decimal place).

REMINDER: reviewDate is not passed into this function, you will set that to be the current date when the review is added. It MUST be in mm/dd/yyyy format.

async getAllReviews(movieId);

This function will return an array of objects of the reviews given the `movieId` return ONLY the reviews for the movie, not any of the other movie data. **If there are no reviews for the movie, this function will return an empty array**

If the `movieId` is not provided, the method should throw.

If the `movieId` provided is not a string, or is an empty string, the method should throw.

If the `movieId` provided is not a valid `ObjectId`, the method should throw

If the movie doesn't exist with that `movieId`, the method should throw.

async getReview(reviewId);

When given a `reviewId`, this function will return a review from the movie. Return ONLY the review object and not all of the movie data.

If the `movieId` is not provided, the method should throw.

If the `movieId` provided is not a string, or is an empty string, the method should throw.

If the `movieId` provided is not a valid `ObjectId`, the method should throw

If the review doesn't exists with that `reviewId`, the method should throw.

```
{ _id: "603d992b919a503b9afb856e",
 reviewTitle: "Meh...",
reviewDate: "10/13/2022",
 reviewerName: "Patrick Hill",
 review: "This movie was good. It was entertaining, but as someone who works in IT, it was not very realisti
c",
 rating: 3.5
}
```

async removeReview(reviewId):

This function will remove the review from the movie in the database and then return the movie object that the review belonged to show that the review sub-document was removed from the movie document.

If the `reviewId` is not provided, the method should throw.

If the `reviewId` provided is not a string, or is an empty string, the method should throw.

If the `reviewId` provided is not a valid `ObjectId`, the method should throw

If the review doesn't exists with that `reviewId`, the method should throw.

routes/movies.js

You must do FULL error handling and input checking for ALL routes! checking if input is supplied, correct type, range etc. and responding with proper status codes when you encounter bad input. All the input types, values and ranges that apply to the DB functions apply to the routes as well so you will do all the same checks in the routes that you do in the DB functions before sending the data to the DB function

`GET /movies`

Responds with an array of all movies in the format of `{"_id": "movie_id", "title": "movie_name"}` Note: Notice you are **ONLY** returning the movie ID, and movie name

```
[{ "_id": "603d965568567f396ca44a72","title": "Hackers"},{ "_id": "704f456673467g306fc44c34","title": "The Br
eakfast Club"},.....]
```

`POST /movies`

Creates a movie with the supplied data in the request body, and returns the new movies **(ALL FIELDS MUST BE PRESENT AND CORRECT TYPE).**

You should expect the following JSON to be submitted in the request.body:

```
{
    title: "Hackers",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
```

```
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min"
}
```

If `title, plot, genres, rating, studio, director, castMembers, dateReleased, runtime` are not provided at all, the route should issue a 400 status code and end the request. (**All fields need to have valid values**);

If `title, plot, rating, studio, director, dateReleased, runtime` are not `strings` or are empty strings, the route should issue a 400 status code and end the request.

If `genres` is not an array that has at least one string element contained in it, the route should issue a 400 status code and end the request.

if any of the elements in `genres` is not a valid string (empty strings or strings with just spaces are invalid), the route should issue a 400 status code and end the request.

If `castMembers` is not an array that has at least one string element contained in it, the route should issue a 400 status code and end the request.

if any of the elements in `castMembers` is not a valid string (empty strings or strings with just spaces are invalid), the route should issue a 400 status code and end the request.

If `dateReleased` is not a valid date string (09/31/2019 is not valid as there are not 31 days in September. 02/30/2020 is not valid as there are not 30 days in February this field MUST be a valid date), or if `dateReleased` is less than 01/01/1900 or greater than the current year + 2 (2024 in this case) the route should issue a 400 status code and end the request.

You do not have to take leap years into account and the format of the date must be in mm/dd/yyyy format. Note: so only years 1900-2024 are valid values. Do not hardcode the year to be 2024, use the current year and then add 2 years to it. This will ensure that the application will function past 2024).

`runtime` MUST be in the following format "#h #min". both #'s must be a positive whole number, but for minutes, that may be zero but the max value for min should be 59 since 60min would be 1 hour:

For example: valid: "2h 30min", "2h 0min", "1h 59min" not valid: "-5h 20min", "3.5h 10min", "0h 30min" (most movies are longer than an 30 min, and usually longer than 1 hour), "2h 60min" (this should just be 3 hours). This field will be case sensitive so you MUST match the format shown exactly. If it is not in this format, the route should issue a 400 status code and end the request.

NOTE: as a reminder, `overallRating` and `reviews` are not passed into this route or the PUT route, When a movie is created, in your DB function, you will initialize the reviews array to be an empty array. You will also initialize overallRating to be 0 when a movie is created.

For PUT: you will not modify the overallRating or reviews properties in the PUT route, you must keep them intact as they were before the update.

If the JSON provided does not match the above schema or fails the conditions listed above, you will issue a 400 status code and end the request.

If the JSON is valid and the movie can be created successfully, you will return the newly created movie (as shown below) with a 200 status code.

```
{
    _id: "507f1f77bcf86cd799439011"),
    title: "Hackers",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
```

```
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[],
    overallRating: 0
}
```

`GET /movies/:movieId`

Example: `GET /movies/507f1f77bcf86cd799439011`

Responds with the full content of the specified movie. So you will return all details of the movie.

if `_id` is not a valid `ObjectId`, you will issue a 400 status code and end the request

If no movie with that `_id` is found, you will issue a 404 status code and end the request.

You will return the movie (as shown below) with a 200 status code along with the movie data if found.

```
{
    _id: "507f1f77bcf86cd799439011"),
    title: "Hackers",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[{_id:"607f1f77bcf86cd799439022",reviewTitle: "Meh...",reviewDate: "10/13/2022", reviewerName: "P
atrick Hill",review: "This movie was good. It was entertaining, but as someone who works in IT, it was not ve
ry realistic", rating: 3.5}],
    overallRating: 3.5
}
```

`PUT /movies/:movieId`

Example: `PUT /movies/507f1f77bcf86cd799439011`

This request will update a movie with information provided from the PUT body. Updates the specified movie **by replacing** the movie with the new movie content, and returns the updated movie. **(All fields need to be supplied in the request.body, even if you are not updating all fields)**

You should expect the following JSON to be submitted:

```
{
    title: "Hackers",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min"
}
```

if the `id` url parameter is not a valid `ObjectId`, you will issue a 400 status code and end the request

If no movie exists with the :id provided, return a 404 and end the request.

If `title, plot, genres, rating, studio, director, castMembers, dateReleased, runtime` are not provided at all, the route should issue a 400 status code and end the request. (**All fields need to have valid values**);

If `title, plot, rating, studio, director, dateReleased, runtime` are not `strings` or are empty strings, the route should issue a 400 status code and end the request.

If `genres` is not an array that has at least one string element contained in it, the route should issue a 400 status code and end the request.

if any of the elements in `genres` is not a valid string (empty strings or strings with just spaces are invalid), the route should issue a 400 status code and end the request.

If `castMembers` is not an array that has at least one string element contained in it, the route should issue a 400 status code and end the request.

if any of the elements in `castMembers` is not a valid string (empty strings or strings with just spaces are invalid), the route should issue a 400 status code and end the request.

If `dateReleased` is not a valid date string (09/31/2019 is not valid as there are not 31 days in September. 02/30/2020 is not valid as there are not 30 days in February this field MUST be a valid date), or if `dateReleased` is less than 01/01/1900 or greater than the current year + 2 (2024 in this case) the route should issue a 400 status code and end the request.

You do not have to take leap years into account and the format of the date must be in mm/dd/yyyy format. Note: so only years 1900-2024 are valid values. Do not hardcode the year to be 2024, use the current year and then add 2 years to it. This will ensure that the application will function past 2024).

`runtime` MUST be in the following format "#h #min". both #'s must be a positive whole number, but for minutes, that may be zero but the max value for min should be 59 since 60min would be 1 hour:

For example: valid: "2h 30min", "2h 0min", "1h 59min" not valid: "-5h 20min", "3.5h 10min", "0h 30min" (most movies are longer than an 30 min, and usually longer than 1 hour), "2h 60min" (this should just be 3 hours). This field will be case sensitive so you MUST match the format shown exactly. If it is not in this format, the route should issue a 400 status code and end the request.

If the JSON provided does not match the above schema or fails the conditions listed above, you will issue a 400 status code and end the request.

NOTE: reviews should not be able to be modified in this route. You must copy the array of review objects from the existing movie first and then insert them into the updated document so they are retained and not overwritten. You should also not modify or overwrite the overallRating field, so keep that field intact as it was before the update.

If the JSON provided in the PUT body is not as stated above or fails any of the above conditions, fail the request with a 400 error and end the request.

If the update was successful, then respond with that updated movie (as shown below) with a 200 status code

```
{
    _id: "507f1f77bcf86cd799439011"),
    title: "Hackers WAS HaCkEd!",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[{_id:"607f1f77bcf86cd799439022",reviewTitle: "Meh...",reviewDate: "10/13/2022", reviewerName: "P
atrick Hill",review: "This movie was good. It was entertaining, but as someone who works in IT, it was not ve
ry realistic", rating: 3.5}],
    overallRating: 3.5
}
```

`DELETE /movie/:movieId`

Example: `DELETE /movies/507f1f77bcf86cd799439011`

if the `id` url parameter is not a valid `ObjectId`, you will issue a 400 status code and end the request

If no movie exists with the provided id url parameter, return a 404 and end the request.

Deletes the movie, sends a status code 200 and returns:

```
{"movieId": "507f1f77bcf86cd799439011", "deleted": true}
```

routes/reviews.js

You must do FULL error handling and input checking for ALL routes! checking if input is supplied, correct type, range etc. and responding with proper status codes when you encounter bad input. All the input types, values and ranges that apply to the DB functions apply to the routes as well so you will do all the same checks in the routes that you do in the DB functions before sending the data to the DB function

`GET /reviews/:movieId`

Example: `GET /reviews/306f1f77bcf86cd799431423`

Getting this route will return an array of all reviews in the system for the specified movie id.

if the `movieId` is not a valid `ObjectId`, you will issue a 400 status code and end the request

If no reviews for the `movieId` are found, you will issue a 404 status code and end the request.

if the `movieId` is not found in the system, you will issue a 404 status code and end the request

You will return the array of reviews (as shown below) with a 200 status code along with the review data if found. Note: the `{....}` is just to show there are more review objects in the array Obviously you should not be returning "`{....}`" literally

```
[{ _id: "603d992b919a503b9afb856e", reviewTitle: "Meh...",

reviewDate: "10/13/2022", reviewerName: "Patrick Hill",

review: "This movie was good. It was entertaining, but as someone who works in IT, it was not very realisti
c",

rating: 3.5 }, {.....}, {....}
]
```

`POST /reviews/:movieId`

Example: `POST /reviews/306f1f77bcf86cd799431423`

Creates a review sub-document with the supplied data in the request body, and returns all the movie data showing the reviews **(ALL FIELDS MUST BE PRESENT AND CORRECT TYPE).**

You should expect the following JSON to be submitted in the request.body:

```
{

reviewTitle: "Meh...",
reviewerName: "Patrick Hill",
review: "This movie was good. It was entertaining, but as someone who works in IT, it was not very realisti
c",
rating: 3.5

}
```

If `reviewTitle, reviewerName, review, rating` are not provided at all, the route will respond with a status code of 400 and end the request. (**All fields need to have valid values**);

If `reviewTitle, reviewerName, review` are not `strings` or are empty strings, the route will respond with a status code of 400 and end the request.

If the `movieId` URL parameter provided is not a valid `ObjectId`, the route will respond with a status code of 400 and end the request.

If the movie doesn't exist with that `movieId`, the route will respond with a status code of 404 and end the request.

If `rating` is not a number from 1 to 5, the route will respond with a status code of 400 and end the request. (floats will be accepted as long as they are in the range 1.5 or 4.8 for example. We will only use one decimal place).

REMINDER: reviewDate is not passed into this route from the request body, you will set that to be the current date when the review is added.

If the JSON provided does not match that schema above or it fails any of the above conditions, you will issue a 400 status code and end the request.

If the JSON is valid and the review can be created successful, you will return all the movie data showing the reviews (as shown below) with a 200 status code.

```
{
    _id: "507f1f77bcf86cd799439011"),
    title: "Hackers WAS HaCkEd!",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[{_id:"607f1f77bcf86cd799439022",reviewTitle: "Meh...",reviewDate: "10/13/2022", reviewerName: "P
atrick Hill",review: "This movie was good. It was entertaining, but as someone who works in IT, it was not ve
ry realistic", rating: 3.5}],
    overallRating: 3.5
}
```

`GET /reviews/review/:reviewId`

Example: `GET /reviews/review/603d992b919a503b9afb856e`

if the `reviewId` is not a valid `ObjectId`, you will issue a 400 status code and end the request

If no review for the `reviewId` are found, you will issue a 404 status code and end the request.

You will return the review (as shown below) with a 200 status code.

```
{
  _id: "603d992b919a503b9afb856e",
  reviewTitle: "Meh...",
  reviewDate: "10/13/2022",
  reviewerName: "Patrick Hill",
  review: "This movie was good.  It was entertaining, but as someone who works in IT, it was not very realist
ic",
  rating: 3.5
}
```

`Delete /reviews/review/:reviewId`

Example: `DELETE /reviews/review/603d992b919a503b9afb856e`

Deletes the specified review for the specified `reviewId`, and then sends a 200 status code and returns the full movie data that the review belonged to to show the review has been removed (The below example only had one review so the review has been removed and overallRating set back to be 0):

```
{
    _id: "507f1f77bcf86cd799439011"),
    title: "Hackers WAS HaCkEd!",
    plot: "Hackers are blamed for making a virus that will capsize five oil tankers.",
    genres: ["Crime", "Drama", "Romance"],
    rating: "PG-13",
    studio: "United Artists",
    director: "Iain Softley",
    castMembers: ["Jonny Miller", "Angelina Jolie", "Matthew Lillard", "Fisher Stevens"],
    dateReleased: "09/15/1995",
    runtime: "1h 45min",
    reviews:[],
    overallRating: 0
}
```

if the `reviewId` url parameter is not a valid `ObjectId`, you will issue a 400 status code and end the request.

If no review with that `reviewId` is found, you will issue a 404 status code and end the request.

NOTE: If a review is deleted, you need to recalculate the average for the `overallRating` field in the main movie document

app.js

Your app.js file will start the express server on **port 3000**, and will print a message to the terminal once the server is started.

Tip for testing:

You should create a seed file that populates your DB with initial data for both movies and reviews. This will GREATLY improve your debugging as you should have enough sample data to do proper testing and it would be rather time consuming to enter a movie and reviews for that movie one by one through the API. A seed file is not required and is optional but is highly recommended. You should have a DB with at least 10 movies and multiple reviews for each movie for proper testing (again, this is not required, but it is to ensure you can test thoroughly.)

General Requirements

1. You **must not submit** your node_modules folder
2. You **must remember** to save your dependencies to your package.json folder
3. You must do basic error checking in each function
4. Check for arguments existing and of proper type.
5. Throw if anything is out of bounds (ie, trying to perform an incalculable math operation or accessing data that does not exist)
6. If a function should return a promise, you should mark the method as an `async` function and return the value. Any promises you use inside of that, you should *await* to get their result values. If the promise should throw, then you should throw inside of that promise in order to return a rejected promise automatically. Thrown exceptions will bubble up from any awaited call that throws as well, unless they are caught in the async method.
7. You **must remember** to update your package.json file to set `app.js` as your starting script!
8. You **must** submit a zip file named in the following format: `LastName_FirstName_CS546_SECTION.zip`