Lab 3

Due Oct 16, 2023 by 11:59pm **Points** 100 **Submitting** a file upload

CS-554 Lab 3

Creating a GraphQL Server

For this assignment, We are going to create a GraphQL server using Apollo Server. We will also be using Redis and MongoDB!

Database Schema

For the MongoDB data, you will be using two collections: `authors` and `books`. The schema for them is as follows:

Authors:

```
{
  _id: UUID,
  first_name: string,
  last_name: string,
  date_of_birth: string,
  hometownCity: string,
  hometownState: string,
  books: [array_of_book_ids](relates to book _id's from the books collection)
}
```

Books:

```
{
  _id: UUID,
  title: string,
  genres: [array_of_strings],
  publicationDate: string,
  publisher: string,
  summary: string,
  isbn: string,
  language: string,
  pageCount: number,
  price: number,
  format: [array_of_strings],
  authorId: UUID (corresponds to an author's ID in authors.json
}
```

You should seed your database with some initial data. We have provided you with a seed file which you can download below. There are 1000 books and 500 authors in the seed file. Every book has an author ID that corresponds with an author, but not every author has a book/books in their books array. So you can play around with adding some books that don't have authors if you want. You should load this data into MongoDB only as seed data! You should still test out ALL your queries and mutations!

- **seed.js (https://sit.instructure.com/courses/68884/files/12175907?wrap=1) (https://sit.instructure.com/courses/68884/files/12175907/download?download_frd=1) (https://gist.githubusercontent.com/graffixnyc/3381b3ba73c249bfcab1e44d836acb48/raw/e14678cd750a4c4a93**

Schema

We are going to have two main types for our data (excluding queries and mutations): `Books` and `Authors`.

```
type Book {
  _id: String,
  title: String,
  genres: [String],
  publicationDate: String,
  publisher: String,
  summary: String,
  isbn: String,
  language: String,
  pageCount: Int,
  price: Float,
  format: [String],
  author: Author #We will need a resolver for this one!
}

type Author {
  _id: String,
  first_name: String,
  last_name: String,
  date_of_birth: String,
  hometownCity: String,
  hometownState: String,
  numOfBooks: Int #We will need a resolver for this one! This is a computed field that will count the number of boo
ks the author has written (see lecture code for numOfEmployees on the employer type)
  books(limit: Int): [Book]  #We will need a resolver for this one! the limit param is optional, if it's supplied,
you will limit the results to the number supplied, if no limit parameter is supplied, then you return all the books
for that author
}
```

Queries

1. `authors: [Author]` //This query returns an array of all authors. If the query is cached in Redis, return it from the cache. Otherwise, cache it with **a one hour expire time**.
2. `books: [Book]` //This query returns an array of all books. If the query is cached in Redis, return it from the cache. Otherwise, cache it with **a one hour expire time**.
3. `getAuthorById(_id: String!): Author` //This query gets a single author by ID. If the query is cached in Redis, return it from the cache. Otherwise, cache it with **no** expire time.
4. `getBookById(_id: String!): Book` //This query gets a single book by ID. If the query is cached in Redis, return it from the cache. Otherwise, cache it with **no** expire time.
5. `booksByGenre (genre: String!): [Book]` //This query returns all the books that match the genre supplied. If the query results are cached in Redis, return it from the cache. Otherwise, cache it with **a one hour expire time**. `genre` should be case insensitive meaning searching "HORROR" and "horror" should give you the same results. If the query results for the `genre` are cached in Redis, return the results from the cache. Otherwise, query MongoDB and store the results for that `genre` in the cache using the `genre` as the key. (hint, look at the redis lecture code example full page caching, I keep the searchTerms stored in the redis list all lowercase so it doesn't cache different results for "HORROR" and "horror" and stores the results of either term as "horror" So for example, when you store the results for "HORROR" in redis, you should use

a lowercase variation of the genre "horror" for the Redis key then any time you deal with the genre, refer to the lower case version for the key). Cache it with **a one hour expire time**.

6. **Make sure in your resolver that you check to make sure genre is not just an empty string or string with just spaces.**
7. `booksByPriceRange (min: Float!, max: Float!) : [Book]` //This query returns all the books within a min/max price range (inclusive of the min/max value). If the query is cached in Redis, return it from the cache. Otherwise, cache it with **a one hour expire time**. **For this,** `min` **MUST be a float or whole number >= 0 .** `max` **MUST be a float or whole number greater than the** `min` **value**
8. `searchAuthorsByName (searchTerm: String!): [Author]` //This query will return an array of authors whose first and last name contains the searchTerm provided. The search term can appear ANYWHERE in either the first_name or last_name fields and should be case insensitive meaning searching "TOM" and "tom" should give you the same results. If the query results for the searchTerm are cached in Redis, return the results from the cache. Otherwise, query MongoDB and store the results for that searchTerm in the cache using the searchTerm as the key. (hint, look at the redis lecture code example full page caching, I keep the searchTerms stored in the redis list all lowercase so it doesn't cache different results for "TOM" and "tom" and stores the results of either term as "tom" So for example, when you store the results for "TOM" in redis, you should use a lowercase variation of the searchTerm "tom" for the Redis key then any time you deal with the searchTerm, refer to the lower case version for the key). Cache it with **a one hour expire time**. **Make sure in your resolver that you check to make sure searchTerm is not just an empty string or string with just spaces!**

Remember: When we use the "!" after the type for the input params (`searchTerm: String!`), that means that the field is required!

Computed Fields/Relationships

Relationships:

`Book`

For the `Book` type, we have `author`, this is a related field, so, we need to set up the resolver for that to link. So in the resolver, the `parentValue` is the book we are looking at. So for the book, we need to find the author who wrote that book. So we need to find the author where their _id is the same as the `parentValue.authorId` field.

`Author`

For the `Author` type, we need to set up the relationship between an author's `_id` and the books that have that author `_id` as the value in their `authorId` field. Here the `parentValue` is the `author`, so we will query the `Book` type to get all books by that author. **This resolver, will take in an optional** `limit` **input param. If the** `limit` **param is supplied, you will limit the number of books that are returned for that author by that number. If the limit param is supplied, it MUST be a whole number greater than 0 If the** `limit` **param is not supplied, they you would return ALL books by that author**

Computed Field:

We only have one computed field in our `Author` type, `numOfBooks`. This will return a count of all the books that author has written.

Mutations

1. `addAuthor(first_name: String!, last_name: String!, date_of_birth: String!, hometownCity: String!, hometownState: String!) -> Author`: This mutation will create an `Author` and will save the `Author` in MongoDB. Outside of the required fields, by default, the following values of `Author` should be as they are below. Once the `Author` is added to MongoDB, add them to the Redis cache. Make sure that `numOfBooks` is properly calculated! **You must make sure that all string input params are not just strings with spaces. For state, it should only allow a two letter state abbreviation "NJ" and should be a VALID state. `first_name` and `last_name` should only contain letters A-Z (all cases) and no numbers! Notice the books array is not passed in. When an author is created, you will initialize `books` to be an empty array, and use `uuid` to generate the `_id` field**
2. `editAuthor(_id: String!, first_name: String, last_name: String, date_of_birth: String, hometownCity: String, hometownState: String) -> Author`: This mutation will take care of any updates that we want to perform on a particular `Author`. This mutation should work like a PATCH request, meaning that **one or more** of the optional fields need to be provided. The mutation should not allow someone to edit the `books` array. When the `Author` is edited, make sure that the Redis cache is also updated. **You must make sure that all string input params are not just strings with spaces. For state, it should only allow a two letter state abbreviation "NJ" and should be a VALID state. You should also validate the `date_of_birth` field to make sure a VALID date is passed in. "9/31/2023" is not a valid date since there are not 31 days in September.**
3. `removeAuthor(_id: String!) -> Author`: Delete an `Author` from the MongoDB collection and remove them from the Redis cache. You MUST also delete ALL books that the deleted author has written. Also make sure the author and their books are removed from the cache if they are stored in the cache.
4. `addBook(title: String!, genres: [String!]!, publicationDate: String!, publisher: String!, summary: String!, isbn: String!, language: String!, pageCount: Int!, price: Float!, format: [String!]!, authorId: String!) -> Book`: This mutation will create a `Book` and will save the `Book` in MongoDB. You need to check whether the `authorId` is a valid `authorId` before adding the `book` (meaning they exist in the DB). Once the `Book` is added to MongoDB, add it to the Redis cache. **You must make sure that all string input params are not just strings with spaces (this includes elements in the genres and format arrays). You should also validate the `publicationDate` field to make sure a VALID date is passed in. "9/31/2023" is not a valid date since there are not 31 days in September. Price should be a whole number or float and should be greater than 0. isbn should follow the isbn format. It should accept either a ISBN-10 or ISBN-13 format! pageCount should be a positive whole number greater than 0**
5. `editBook(_id: String!, title: String, genres: [String], publicationDate: String, publisher: String, summary: String, isbn: String, language: String, pageCount: Int, price: Float, format: [String], authorId: String) -> Book`: This mutation will take care of any updates that we want to perform on a particular `Book`. This mutation should work like a PATCH request, meaning that **one or more** of the optional fields need to be provided. If the `authorId` is updated, then you need to check whether the new `authorId` is a valid `authorId` before editing the `Book` (meaning they exist in the DB). If the author of the book changes, then you MUST remove that book ID from the old author's array of book ID's AND you must push the book ID into the array

of books for the new author. When the `Book` is edited, make sure that the Redis cache is also updated. For updating the author cache, It is probably easiest to just remove the old and new author entries from cache (if they are stored in cache) so the next time, the new updated data is pulled from the DB and put back into cache. **You must make sure that all string input params are not just strings with spaces (this includes elements in the genres and format arrays). You should also validate the `publicationDate` field to make sure a VALID date is passed in. "9/31/2023" is not a valid date since there are not 31 days in September. Price should be a whole number or float and should be greater than 0. isbn should follow the isbn format. It should accept either a ISBN-10 or ISBN-13 format! pageCount should be a positive whole number greater than 0**

6. `removeBook(_id: String!) -> Book`: Delete a `Book` from the MongoDB collection, remove it from the Redis cache, and remove it from the Author's `books` array.

Redis:

For Redis, we will only be caching the data from MongoDB. Meaning, when your resolvers make a query to MongoDB, we are just storing the data returned from MongoDB into the Redis cache. It's very hard to cache GraphQL queries since every query can be different (as mentioned in lecture) so that's why we are just going to cache the data returned from Mongo.

You will cache the whole Books query results (`books: [Book]`) that returns all the books from Mongo. You can store this using any key you like.

You will cache the whole Author query results (authors`: [Author]`) that returns all the Authors from Mongo. You can store this using any key you like.

When you call `getBookById(_id: String!)`, you will store the individual book returned from Mongo in the cache. it would make most sense to use the _id as the key for each individual book in your cache

When you call `getAuthorById(_id: String!)`, you will store the individual author returned from Mongo in the cache. it would make most sense to use the _id as the key for each individual author in your cache.

When you call `booksByGenre (genre: String!)` You will store the results returned from Mongo in a Redis List. You can use the provided genre input param as the key.

When you call `searchAuthorsByName (searchTerm: String!)` You will store the results returned from Mongo in a Redis List. You can use the provided searchTerm input param as the key.

Notes

1. Remember to submit your `package.json` file but **not** your `node_modules` folder.
2. Remember to add the start command!