Project Report - Task 4

Data Storage Paradigms, IV1351

Reza Hosseini rezahos@kth.se

December 2022

1 Introduction

In this task we develop a basic website for Soundgood Music School. The website should be able to handle transactions about instruments and renting them. It should be able to list instruments, rent instruments and terminate rentals.

The higher grade part requires the project to follow the MVC and Layer pattern model. It also needs to have an integration layer DAO that connects the website to the database.

2 Literature Study

In order to do the project we watched the YouTube videos from Leif Lindbäck:

- Introduction to JDBC
- architecture

We went also through the example project "jdbc-bank" to become familiar with the MVC model.

3 Method

For this project we used JavaScript as our main language. We created a NodeJs API using ExpressJS and JavaScript without any front-end interface and the way the website interacts with the user is through the API endpoints which is done by sending GET requests and getting the responds in JSON format.
Since the interaction occurs via JSON responses, we used a chrome-extension called `JSON formatter` that shows the JSON responses much better and it also makes the links in the results clickable which is important because then you can easily navigate through the app otherwise you need to manually go to those links (examples will be shown later in the report).

The project was mainly based on the MVC model and how the user can interact with the website and how the website can interact with the database. The project is basically seperated in 4 parts: View, Controller, Model and Integration. This model is explained more in the Discussion part.

The way project works is: you interact through API endpoints. You can login, see all rented instruments and also you can terminate rentals. All these have their own API points that can be reached by sending GET requests.
The API points then interact with our model thorough controller that in its turn requests data from database.

4 Result

All files for the project is found in the following GitHub repository:
`https://github.com/Rezaavoor/Soundgood-Music/tree/master/Task4`

`Requirements for running the project:`
You need to have a database called `sgm` that have all our data. The file for creating the schema of the database and inserting data into it is found in the `resources` folder in the project directory. The password for the database should be set to "12345" and the user "postgres" should be accessible otherwise you need to change these credentials in the file `DAO.js` according to your own credentials.
You need to have NodeJS installed and by cloning the GitHub repository and running the code `"npm install"` in the project folder, you should have all libraries and dependencies installed.
You can finally run `"npm run dev"` command to start the project and you should be able to go to the localhost:3000 link and see the project running.

In this project all our SQL queries are in ACID form:

- Atomicity: All our queries begin with `BEGIN` statement and end with `COMMIT` statement which guarantees that either all of the operations in the transaction are performed or none of them are.
- Consistency: Our update transaction that does the termination of rentals is consistent because it locks the specific row and ensures that this transaction is the only one accessing and changing the value. This prevents the risk of having dirty writes.
- Isolation: Our transactions are isolated because they are executed as if they were the only transaction running at that time.
- Durability: They are also durable because their changes are persisted after they are the commitment.

An example of how the project is used is shown below:
First the user needs to login which is done by requesting `/students` which shows us all registered students:

Figure 1: JSON response from requesting "/students"

The user logs in by clicking on `linkToProfile` after finding its name. This has obviously security issues but we did it this way to save time because this was not part of the task 4.
Let's assume we are `Ralph Patton` with studentID=2 and we click on our linktoProfile, then we see the following:

Figure 2: JSON response from requesting "/student/2"

Here you can see that Ralph has one rented instrument and its information is also shown. Ralph can rent maximum number of 2 instruments which means `canRentInstrument` is set to true.

Let's now try to rent another instrument by clicking on `linkToRentalPage` and then we see the following:

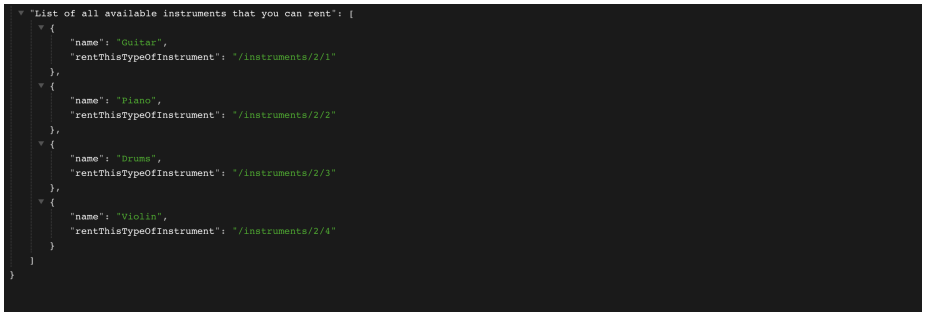

Figure 3: JSON response from requesting "/instruments/2/instrumentTypes"

We want to rent a Violin so we click on `rentThisTypeOfInstrument` from the Violin part and we then can see all available instruments of the type Violin that we can rent:

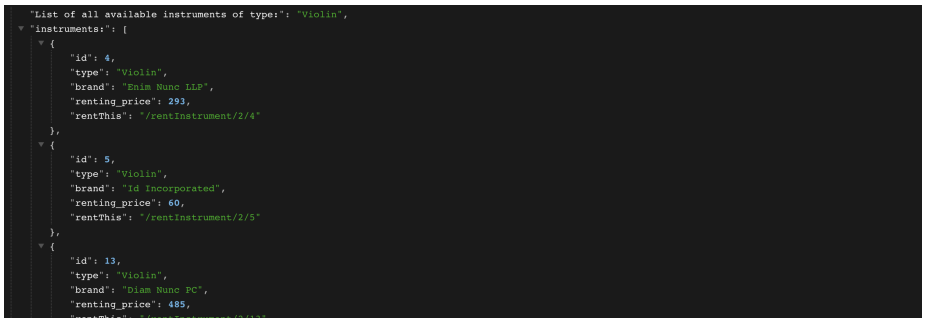

Figure 4: JSON response from requesting "/instruments/2/4"

Let's try to rent the violin from `Id Incorporated` since it's the cheapest one, so we click on its "rentThis" and see the following:

Figure 5: JSON response from requesting "/rentInstrument/2/5"

This shows that the rental is successfully done. We can assure that by going back to our profile by clicking "goToProfile" and we can see that now Ralph has 2 rented instruments and one of the is actually our violin that we just rented:

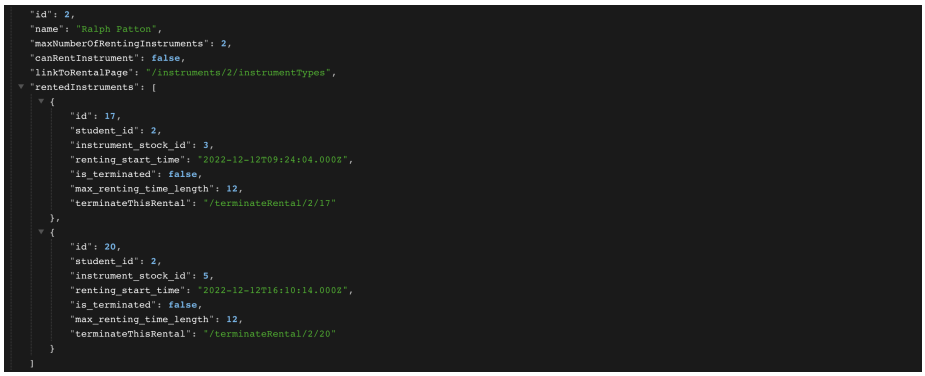

Figure 6: JSON response from requesting "/students/2"

We can also see that Ralph cannot rent anymore instruments since `canRentInstrument` is set to false now.

Let's now try to terminate a rental, we can terminate the rental with id=17 so we click on "terminateThisRental" of the item and see the following:

Figure 7: JSON response from requesting "/terminateRental/2/17"

By going back to profile again we can see that Ralph has now one rented instrument left and `canRentInstrument` is set back to true:

Figure 8: JSON response from requesting "/students/2"

5 Discussion

MVC Model

The entire project is done based on the MVC model. The project is divided into 4 parts: View, Controller, Model, Integration(DAO).

- View: This layer handles the interface of the application which directly interacts with the user. This layer consists of 7 API endpoints and the user can send GET requests to these endpoints and receive results in JSON format. The user is also able to send data via API parameters, for instance "/students/1" sends a GET request to students endpoint with an additional data which is the studentID.
- Controller: This layer receives the user inputs through the View layer and translates them into actions that can be performed by the Model or View layers. This layer "glues" the Model and the View together. For instance if a user requests the view for its information by passing its ID(GET request: /students/ID), the Controller receives that ID from the View and calls a method(getStudentInformation) that asks the Model layer to retrieve related data from the Database(Student.get) via other necessary leyars.
- Model: This layer manages the structure of the data the application uses, it represents the state of the application and also performs operations on the data. The model interacts with the Integration layer to retrieve data and update states. This layer as well as the Controller layer has no direct access to the user input and the View model is their middle-layer.
- Integration(DAO): The Integration layer has direct access to the database and this is where all the SQL queries are found. Typically the DAO is considered part of the Model but technically even Controller is allowed to reach out to the DAO. However in this project, the DAO is exclusively accessed by the Model layer. For instance if a student is asked for its information, a method(Student.get(id)) in the Model gets called which on its turn calls a method from the DAO (getStudentById(id)). Then some SQL queries are run and the data coming from the database is returned to the Model via the DAO.

One benefit of using the MVC model is that it is easier to work on different parts of the application without worrying about other parts. It also makes it easier for multiple developers to simultaneously update different parts.

not using SELECT FOR UPDATE

We ended up not needing to use this query as we never need to read data before updating and writing it back. The only UPDATE implementation we have is where we terminate our rental and that is done by setting `is-terminated` value to `true`. Previously, we toggled the boolean value which required us to read the data first but since we always set the value `false` so we don't necessarily need to read the data.

Termination of a rental

The way we had implemented our model before was that whether a studentId is attached to the the rental or not is what decides if a rental is terminated or not. If a rental lacks studentId that means that the student is no longer renting that instrument. This works fine but the problem is that you loose information about the student, you cannot then see who rented and terminated a particular instrument. Therefore we changed our model and added a new column to our table called `is-terminated` which is false when a student rents an instrument, but sets to true when he/she terminates the rental. Doing it this way, we preserve all data even after termination.

Crash and Error handling

There is almost no error handling in our project when it comes to receiving the user data. For instance if a user asks for information of a student with id=9999999, the server crashes. The reason we haven't fixt that is firstly, it wasn't a mandatory part of the task but secondly, the app is not intended to accept manual data. A user is not supposed to be able to request manually but it should happen through the app which is crash-free. As long as a user only interacts with the app via the links given by the app, no error should occur.