Programmer's Manual

1. Project Description:

This project involves the development of a music streaming platform focused on offering a functional experience integrated with database systems. The platform was developed using established software development practices, aiming to create a reliable and scalable system.

The platform supports three types of users: consumers, artists, and administrators. Consumers have access to music tracks, with additional features available to premium subscribers, such as creating and managing playlists. Artists can publish songs and albums, while administrators are responsible for user management and issuing prepaid cards for premium subscriptions.

The main features include user registration and authentication, music publishing, playlist management, and subscription administration through prepaid cards. The platform's backend is built on a transactional relational database, accessed via a REST API that supports operations such as GET, POST, and PUT.

The project was developed with a focus on security, efficient transaction and concurrency management, and robust error handling. It is strongly recommended to consult the database tables after performing operations to visualize the effects of these operations, ensuring that transactions and system changes are occurring as expected.

2. Initial Setup:

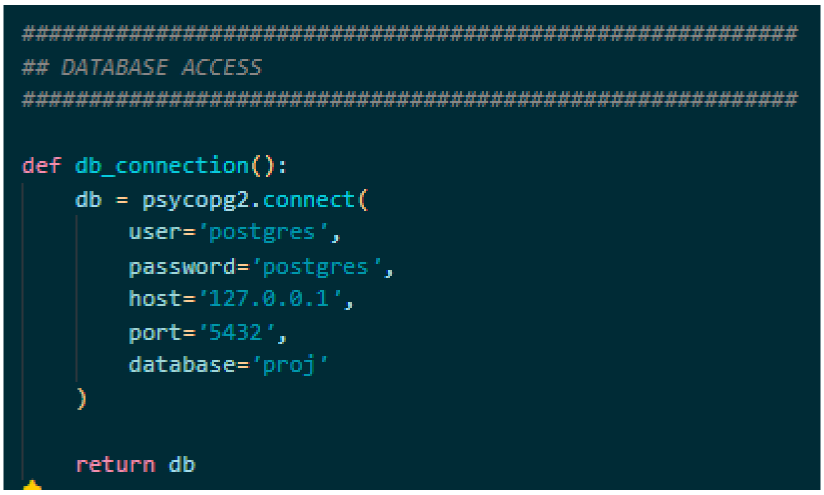
a) **Open pgAdmin4** (if installed) and start a server.

b) **Run the setup script**: In a query tool within pgAdmin4, execute the script “setup.txt” to create the database tables, add essential information, and set up an administrator account. After that, you need to refresh your database on pgAdmin4 to see the tables.

c) **Manual Adjustment Required**: After running the script, a manual change is needed in the music table. Right-click on the table, select "Properties," and change the data type of the duration column to “time without timezone”.

d) **Run the triggers script:** Open another query tool and execute the script contained in the “create_triggers.txt” file. This will create the necessary triggers in the database. The database is now ready to receive information.

e) **Set up the Python environment:** Place the “demo-api.py” code in a Python interpreter or IDE. If you are using a text editor like VSCode, make sure Python 3 is installed on your machine. Before running the program, add the corresponding database credentials where you ran the scripts into the “demo-api.py” file:



```
##########################################################
## DATABASE ACCESS
##########################################################

def db_connection():
    db = psycopg2.connect(
        user='postgres',
        password='postgres',
        host='127.0.0.1',
        port='5432',
        database='proj'
    )

    return db
```

2.1.1 Image

Then, start the server:

python demo-api.py

f) **Open Postman** (if installed on your machine): Use URLs in the format http://localhost:8080/endpoint_name_described_in_the_code to test the developed functionalities.

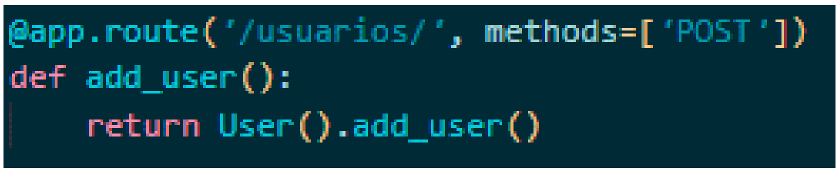
3. User Management:

In this section, we will configure and manage the different types of users on the platform. This includes adding consumers and artists, authenticating as an administrator (including an explanation of the access token), and generating prepaid cards that will be used by consumers for subscription. Practical instructions for each of these steps will be provided below.

○ Add Consumer:

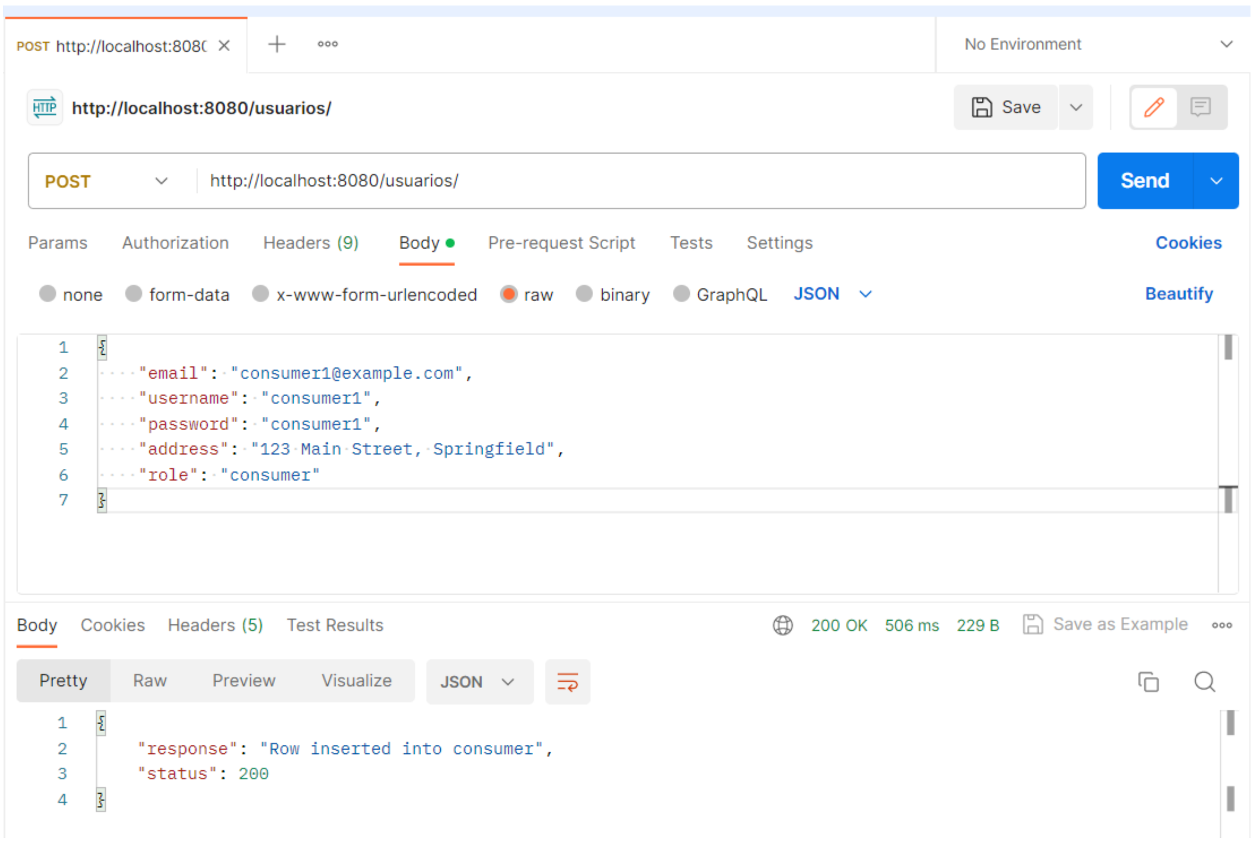
To add a new consumer to the platform, we use a POST endpoint, as shown in the **3.1.1 Image**. The Python code defines a route (/usuarios/) that accepts POST requests. This function calls the add_user() method from the User class, which handles inserting the new consumer's data into the database.

In **3.1.2 Image**, we see an example of a request being made via Postman. The body of the request contains the necessary information to create the consumer, including email, username, password, address, and role. After submitting the request, the server responds with a message confirming that the row was successfully inserted into the consumer table, accompanied by a status code 200, indicating that the operation was successful.



```
@app.route('/usuarios/', methods=['POST'])
def add_user():
    return User().add_user()
```

3.1.1 Image



POST http://localhost:8080/usuarios/

```
{
    "email": "consumer1@example.com",
    "username": "consumer1",
    "password": "consumer1",
    "address": "123 Main Street, Springfield",
    "role": "consumer"
}
```

200 OK 506 ms 229 B

```
{
    "response": "Row inserted into consumer",
    "status": 200
}
```

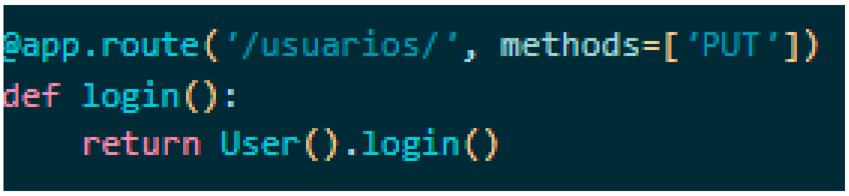
3.1.2 Image

- ## Log in as Administrator and Token Explanation:

  To log in as an administrator on the platform, we use a PUT endpoint, as demonstrated in the **3.2.1 Image**. The Python code defines a route (/usuarios/) that accepts PUT requests and calls the login() method from the User class, which validates the provided credentials and generates an authentication token.

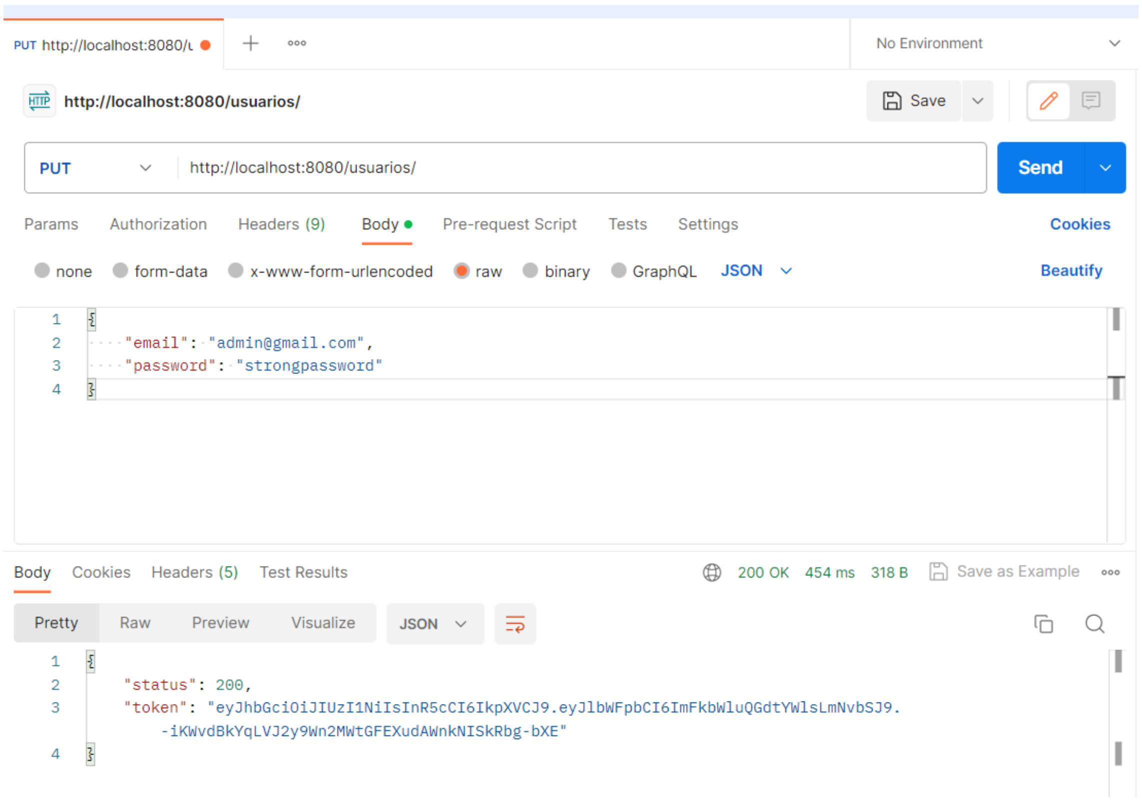
In the **3.2.2 Image**, we see an example of a request being made via Postman. The request body includes the administrator's email and password. After submitting the request, the server responds with a status code 200 and returns an authentication token, as shown in the JSON response.

In the **3.2.3 Image**, the generated token is used to authorize future requests, ensuring that the administrator has the appropriate permissions. The token is added to the request header as a Bearer Token, allowing the system to identify and authorize the user to perform operations that require administrative privileges.



```
@app.route('/usuarios/', methods=['PUT'])
def login():
    return User().login()
```

3.2.1 Image



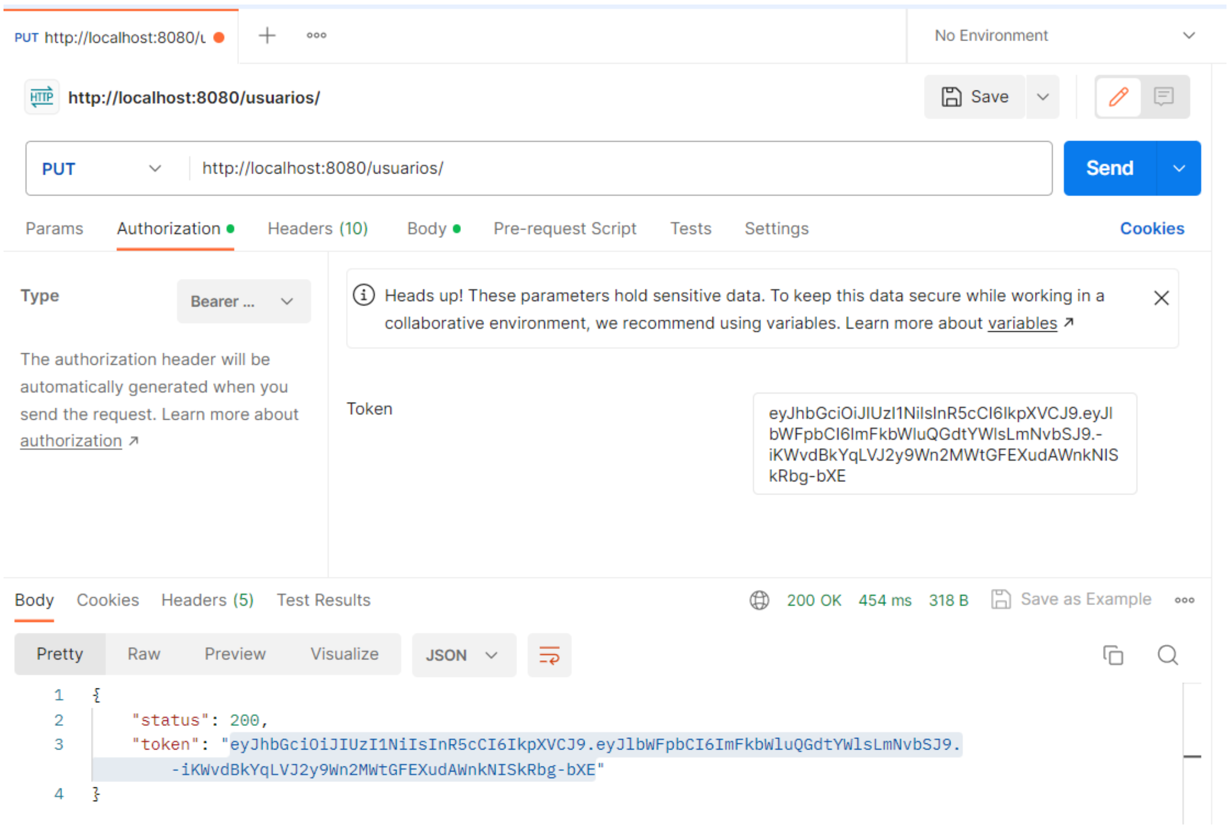
PUT http://localhost:8080/usuarios/

```
{
    "email": "admin@gmail.com",
    "password": "strongpassword"
}
```

200 OK 454 ms 318 B

```
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFkbWluQGdtYWlsLmNvbSJ9.
        -iKWvdBkYqLVJ2y9Wn2MWtGFEXudAWnkNISkRbg-bXE"
}
```

3.2.2 Image



PUT http://localhost:8080/usuarios/

No Environment

PUT http://localhost:8080/usuarios/

Authorization — Type: Bearer ...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFkbWluQGdtYWlsLmNvbSJ9.-iKWvdBkYqLVJ2y9Wn2MWtGFEXudAWnkNISkRbg-bXE

200 OK 454 ms 318 B

```
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFkbWluQGdtYWlsLmNvbSJ9.
        -iKWvdBkYqLVJ2y9Wn2MWtGFEXudAWnkNISkRbg-bXE"
}
```

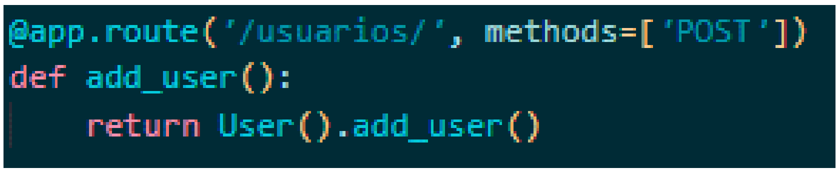
3.2.3 Image

○ Add Artist:

To add a new artist to the platform, we use a POST endpoint, as demonstrated in the **3.3.1 Image** below. The Python code defines a route (/usuarios/) that accepts POST requests and calls the add_user() method from the User class. This method is responsible for inserting the new artist's data into the database.

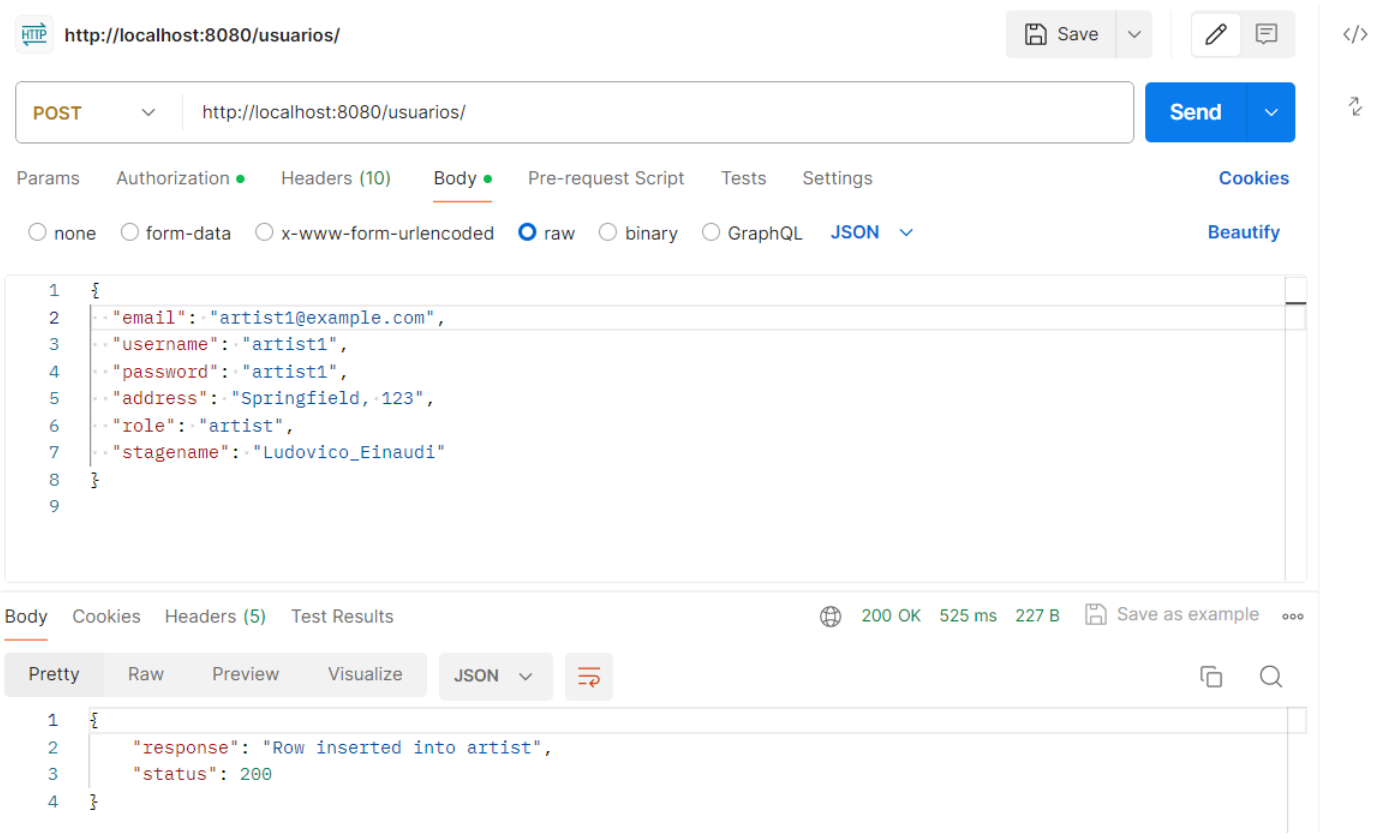
In the **3.3.2 Image**, we see an example of a request being made via Postman. The request body includes relevant artist information such as email, username, password, address, role (in this case, "artist"), and stage name. After submitting the request, the server responds with a status code of 200 and a message confirming that the row was inserted into the artist table, as shown in the JSON response.

This operation ensures that new artists can be registered on the platform, allowing them to access and use the resources designated for them.



```python
@app.route('/usuarios/', methods=['POST'])
def add_user():
    return User().add_user()
```

3.3.1 Image



http://localhost:8080/usuarios/

POST http://localhost:8080/usuarios/

```json
{
  "email": "artist1@example.com",
  "username": "artist1",
  "password": "artist1",
  "address": "Springfield, 123",
  "role": "artist",
  "stagename": "Ludovico_Einaudi"
}
```

200 OK 525 ms 227 B

```json
{
    "response": "Row inserted into artist",
    "status": 200
}
```

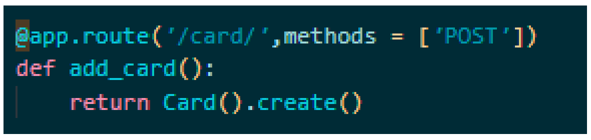
3.3.2 Image

○ Generate Prepaid Cards:

To add new purchase cards to the platform, we use a POST endpoint, as demonstrated in the **3.4.1 Image** below. The Python code defines a route (/card/) that accepts POST requests and calls the create() method from the Card class. This method is responsible for creating the new purchase cards in the database.

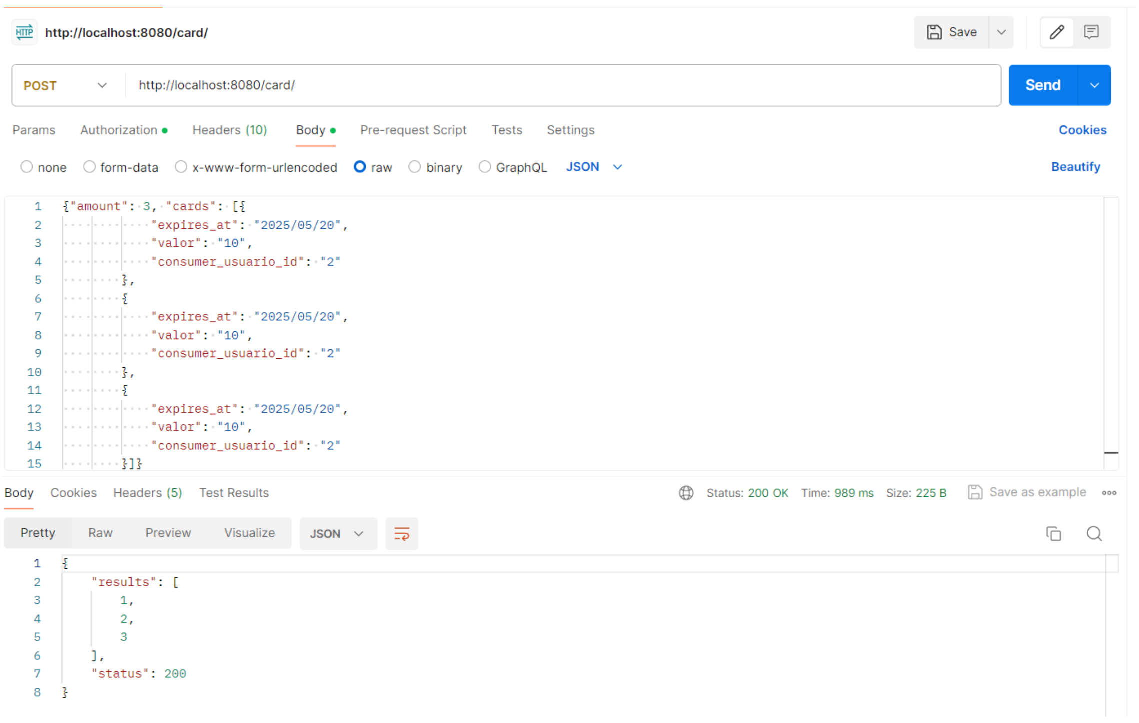
In the **3.4.2 Image**, we see an example of a request being made via Postman. The request body contains information for creating three purchase cards, each with a value of 10 coins and an expiration date, all associated with the newly created consumer. The consumer ID is number 2 because he was the second user created in the system. The first user, with ID 1, was the administrator, created during the setup phase. After submitting the request, the server responds with a status code of 200 and a list of the IDs of the created cards, as shown in the JSON response.

It is essential to store these card IDs, as they will be used to make a subscription purchase. These IDs function as unique identifiers for each card, allowing the system to correctly associate the payment with the desired subscription. Proper management of these IDs ensures that each card can be processed correctly in subscription transactions, maintaining the integrity and control of financial operations.



```
@app.route('/card/',methods = ['POST'])
def add_card():
    return Card().create()
```

3.4.1 Image



POST http://localhost:8080/card/

```
{"amount": 3, "cards": [{
            "expires_at": "2025/05/20",
            "valor": "10",
            "consumer_usuario_id": "2"
        },
        {
            "expires_at": "2025/05/20",
            "valor": "10",
            "consumer_usuario_id": "2"
        },
        {
            "expires_at": "2025/05/20",
            "valor": "10",
            "consumer_usuario_id": "2"
        }]}
```

Status: 200 OK Time: 989 ms Size: 225 B

```
{
    "results": [
        1,
        2,
        3
    ],
    "status": 200
}
```

3.4.2 Image

4. Artist Operations:

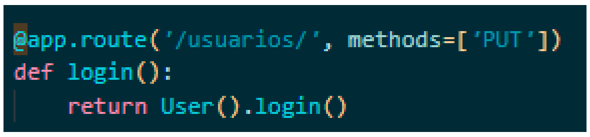
In this section, we will cover the main operations that can be performed by an artist on the platform. The features include the process of logging in as an artist, adding songs, and creating albums. Through these resources, artists will be able to manage their creations directly on the platform, making it easier to publish and organize their musical content.

○ Log in as Artist:

To log in as an artist on the platform, we use a PUT endpoint, as shown in the **4.1.1 Image** below. The Python code defines a route (/usuarios/) that accepts PUT requests and calls the login() method from the User class. This method validates the provided credentials and generates an authentication token.

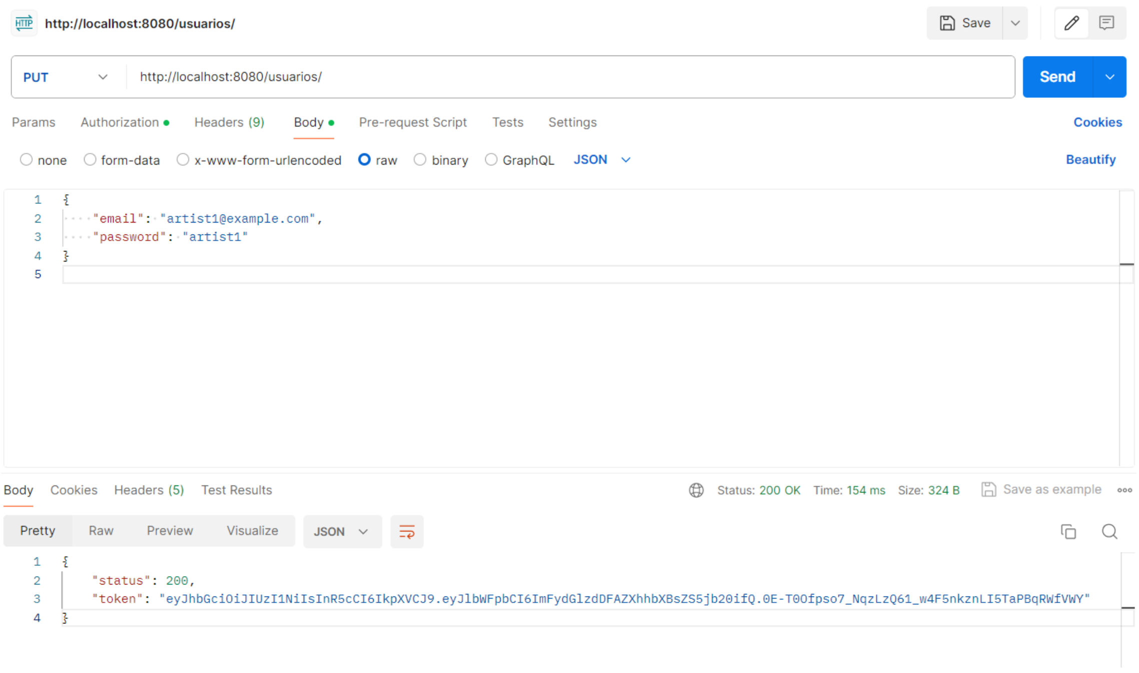
In the **4.1.2 Image**, we see an example of a request being made via Postman. The request body includes the artist's email and password. After submitting the request, the server responds with a status code of 200 and generates an authentication token, as shown in the JSON response.

In the **4.1.3 Image**, the generated token is used to authenticate future requests. It is included in the request header as a **Bearer Token**, allowing the system to identify and authorize the artist to perform specific operations. It is important to securely store the token, as it will be necessary to authenticate subsequent actions on the platform.



```python
@app.route('/usuarios/', methods=['PUT'])
def login():
    return User().login()
```

4.1.1 Image



http://localhost:8080/usuarios/

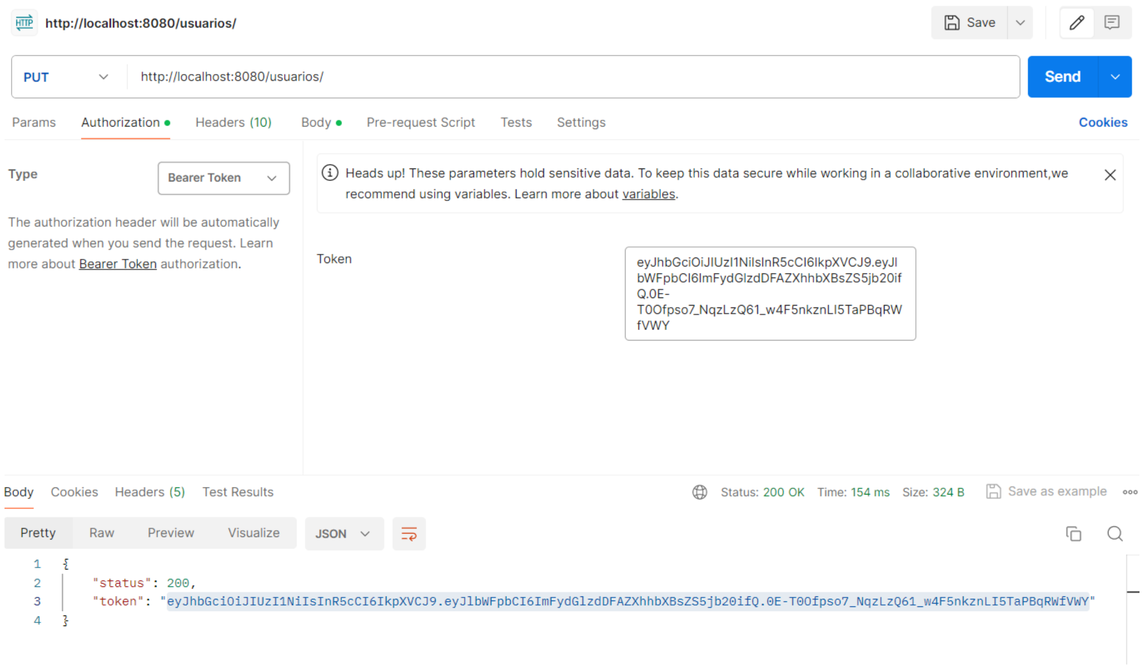
PUT http://localhost:8080/usuarios/

```json
{
    "email": "artist1@example.com",
    "password": "artist1"
}
```

Status: 200 OK Time: 154 ms Size: 324 B

```json
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFydGlzdDFAZXhhbXBsZS5jb20ifQ.0E-T0Ofpso7_NqzLzQ61_w4F5nkznLI5TaPBqRWfVWY"
}
```

4.1.2 Image



http://localhost:8080/usuarios/

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFydGlzdDFAZXhhbXBsZS5jb20ifQ.0E-T0Ofpso7_NqzLzQ61_w4F5nkznLI5TaPBqRWfVWY

Status: 200 OK Time: 154 ms Size: 324 B

```
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFydGlzdDFAZXhhbXBsZS5jb20ifQ.0E-T0Ofpso7_NqzLzQ61_w4F5nkznLI5TaPBqRWfVWY"
}
```

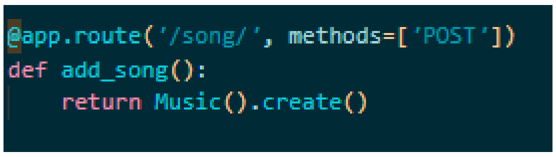
4.1.3 Image

- Add Songs:

To add songs to the platform, we use a POST endpoint, as shown in the **4.2.1 Image** below. The Python code defines a route (/song/) that accepts POST requests and calls the create() method from the Music class. This method is responsible for inserting the new song's data into the database.

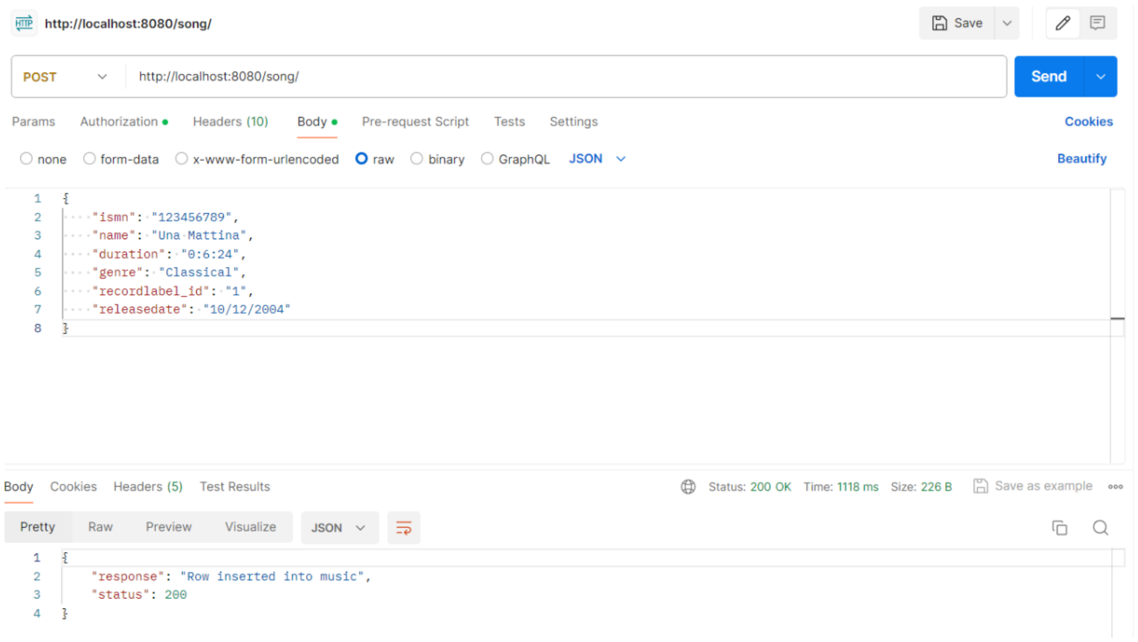
In the **4.2.2 Image** and **4.2.3 Image**, we see examples of requests being made via Postman. The request body includes relevant information about the song, such as the ISMN (International Standard Music Number), the song name, duration, genre, record label ID, and release date. After submitting the request, the server responds with a status code of 200 and confirms that the row was inserted into the songs table, as shown in the JSON response.

This process ensures that the songs are correctly registered on the platform, allowing users to access and interact with them.



```
@app.route('/song/', methods=['POST'])
def add_song():
    return Music().create()
```

4.2.1 Image



http://localhost:8080/song/

POST http://localhost:8080/song/

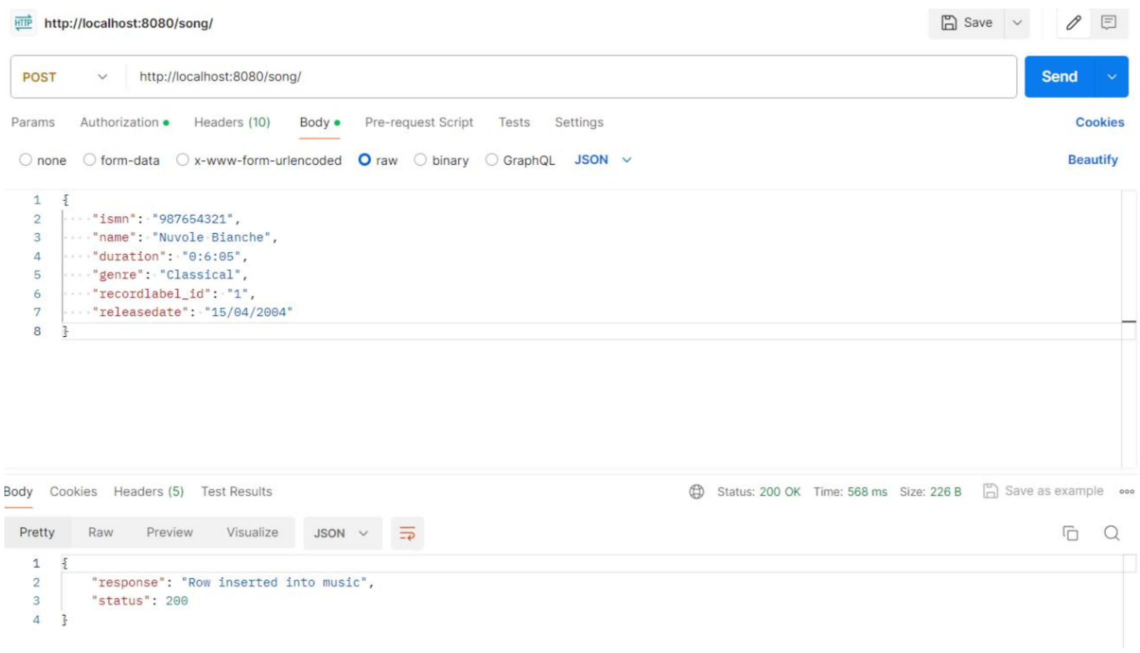
Body: raw, JSON

```
{
    "ismn": "123456789",
    "name": "Una Mattina",
    "duration": "0:6:24",
    "genre": "Classical",
    "recordlabel_id": "1",
    "releasedate": "10/12/2004"
}
```

Status: 200 OK Time: 1118 ms Size: 226 B

```
{
    "response": "Row inserted into music",
    "status": 200
}
```

4.2.2 Image



http://localhost:8080/song/

POST http://localhost:8080/song/

Body: raw, JSON

```
{
    "ismn": "987654321",
    "name": "Nuvole Bianche",
    "duration": "0:6:05",
    "genre": "Classical",
    "recordlabel_id": "1",
    "releasedate": "15/04/2004"
}
```

Status: 200 OK Time: 568 ms Size: 226 B

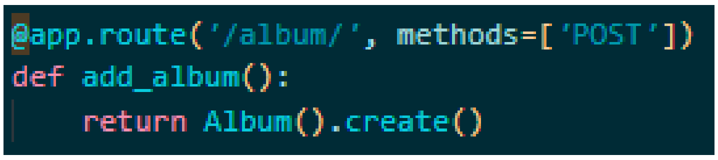
```
{
    "response": "Row inserted into music",
    "status": 200
}
```

4.2.3 Image

○ Add Album:

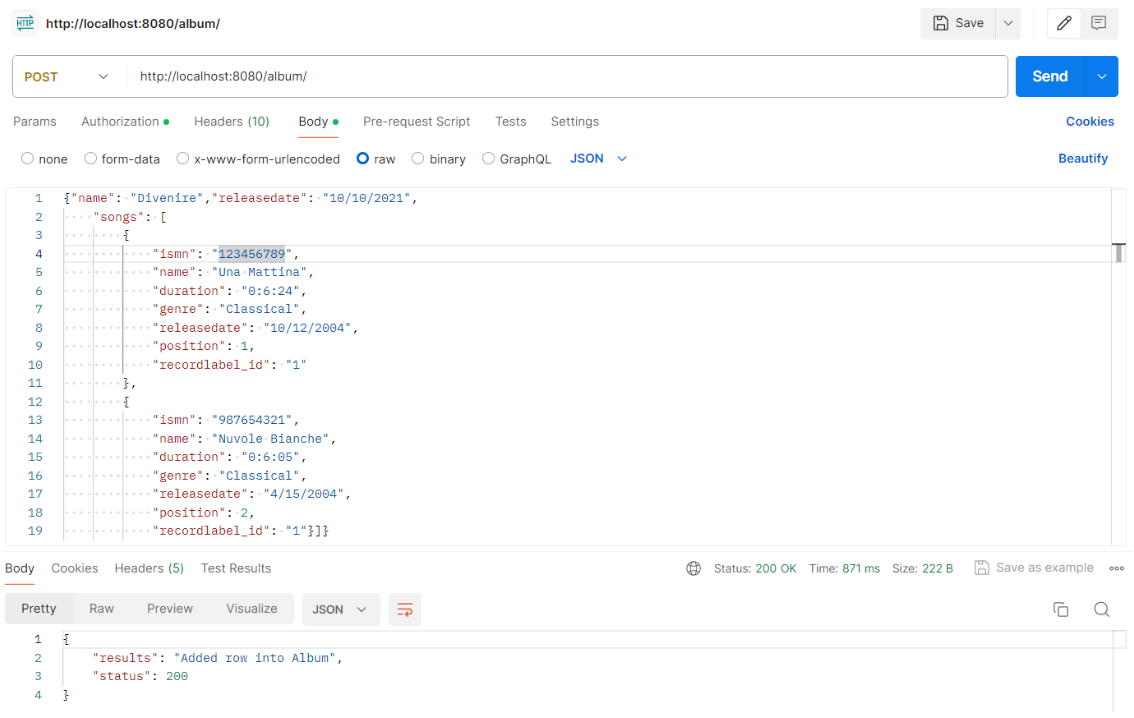
To add an album to the platform, we use a POST endpoint, as shown in the **4.3.1 Image** below. The Python code defines a route (/album/) that accepts POST requests and calls the create() method from the Album class. This method is responsible for inserting the new album's data into the database.

In the **4.3.2 Image**, we see an example of a request being made via Postman. The request body includes the album's information, such as the name, release date, and a list of songs. Each song contains information such as the ISMN, name, duration, genre, position in the album, record label ID, and release date. After submitting the request, the server responds with a status code of 200 and confirms that the album was inserted into the table, as shown in the JSON response.



```python
@app.route('/album/', methods=['POST'])
def add_album():
    return Album().create()
```

4.3.1 Image



```json
{"name": "Divenire","releasedate": "10/10/2021",
    "songs": [
        {
            "ismn": "123456789",
            "name": "Una Mattina",
            "duration": "0:6:24",
            "genre": "Classical",
            "releasedate": "10/12/2004",
            "position": 1,
            "recordlabel_id": "1"
        },
        {
            "ismn": "987654321",
            "name": "Nuvole Bianche",
            "duration": "0:6:05",
            "genre": "Classical",
            "releasedate": "4/15/2004",
            "position": 2,
            "recordlabel_id": "1"}]}
```

```json
{
    "results": "Added row into Album",
    "status": 200
}
```

4.3.2 Image

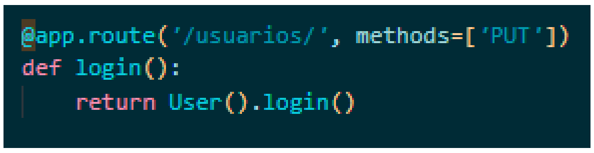
5.Consumer Operations:

○ Log in as Consumer:

To allow a consumer to log in to the platform, we use a PUT endpoint, as shown in the **5.1.1 Image** below. The Python code defines a route (/usuarios/) that accepts PUT requests and calls the login() method from the User class. This method validates the provided credentials and generates an authentication token.

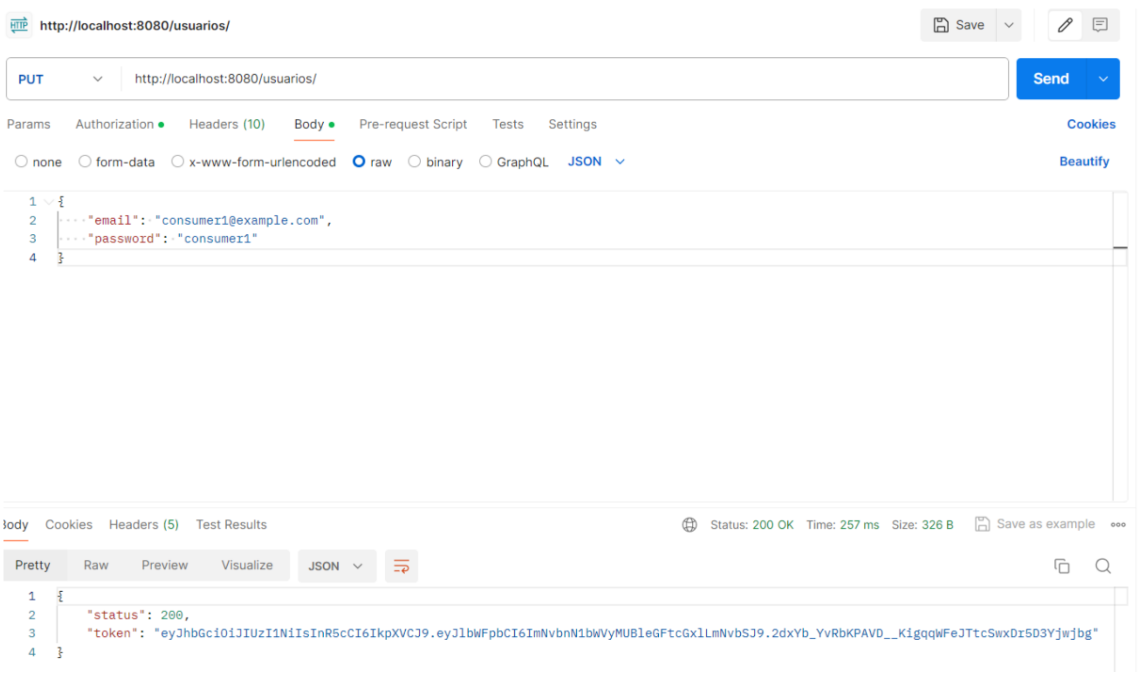
In the **5.1.2 Image**, we see an example of a request being made via Postman. The request body includes the consumer's email and password. After submitting the request, the server responds with a status code of 200 and generates an authentication token, as shown in the JSON response.

In the **5.1.3 Image**, the generated token is used to authenticate future requests. It is included in the request header as a Bearer Token, allowing the system to identify and authorize the consumer to perform specific operations. It is important to store the token securely, as it will be required to authenticate future actions on the platform.



```python
@app.route('/usuarios/', methods=['PUT'])
def login():
    return User().login()
```

5.1.1 Image



PUT http://localhost:8080/usuarios/

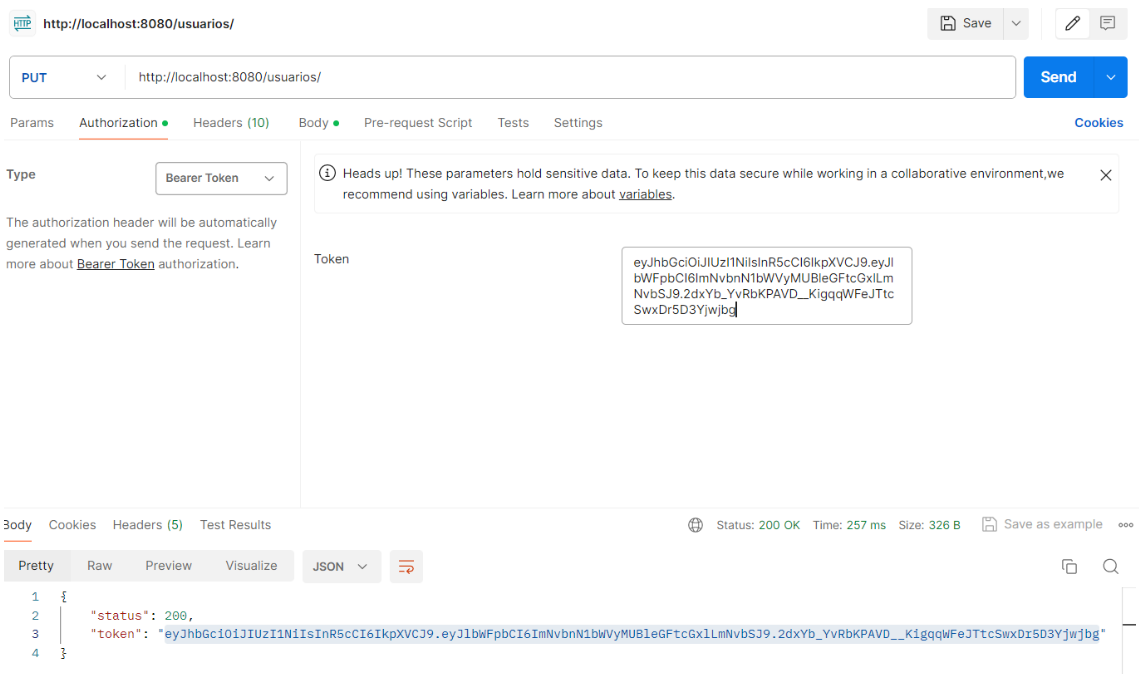
Body: [✓ raw] JSON

```
{
    "email": "consumer1@example.com",
    "password": "consumer1"
}
```

Status: 200 OK Time: 257 ms Size: 326 B

```
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImNvbnN1bWVyMUBleGFtcGxlLmNvbSJ9.2dxYb_YvRbKPAVD__KigqqWFeJTtcSwxDr5D3Yjwjbg"
}
```

5.1.2 Image



PUT http://localhost:8080/usuarios/

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImNvbnN1bWVyMUBleGFtcGxlLmNvbSJ9.2dxYb_YvRbKPAVD__KigqqWFeJTtcSwxDr5D3Yjwjbg

Status: 200 OK Time: 257 ms Size: 326 B

```
{
    "status": 200,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImNvbnN1bWVyMUBleGFtcGxlLmNvbSJ9.2dxYb_YvRbKPAVD__KigqqWFeJTtcSwxDr5D3Yjwjbg"
}
```

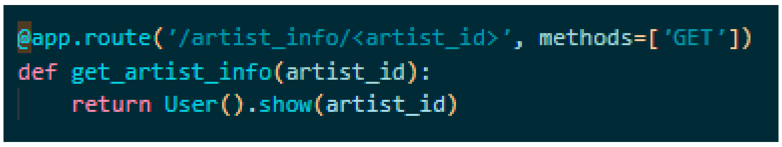
5.1.3 Image

○ Show Artist Information:

To display information about an artist on the platform, we use a GET endpoint, as shown in the **5.2.1 Image** below. The Python code defines a route (/artist_info/<artist_id>) that accepts GET requests and calls the show() method from the User class. This method retrieves the artist's information based on the provided artist_id.

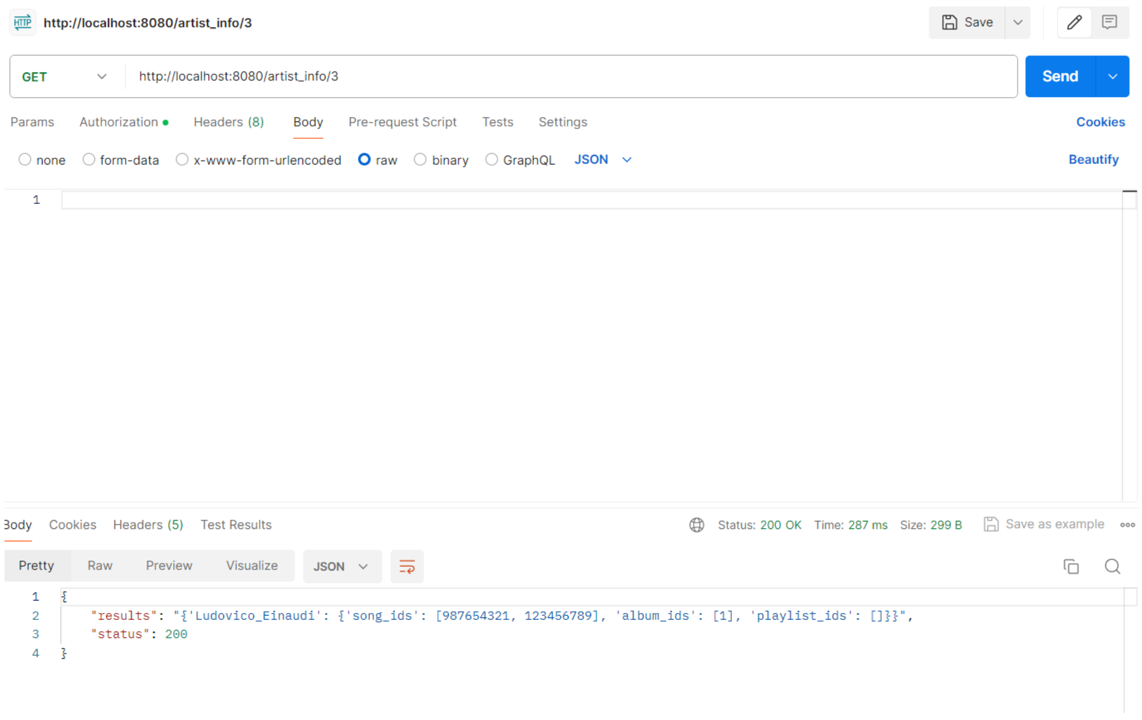
In the **5.2.2 Image**, we see an example of a request being made via Postman for the artist with ID 3. The JSON response includes details such as the artist's name, the IDs of associated songs, album IDs, and linked playlists. The server responds with a status code of 200, confirming that the artist's information was successfully retrieved.

This process allows detailed information about the artist, including their songs and albums, to be easily accessed through the platform.



```
@app.route('/artist_info/<artist_id>', methods=['GET'])
def get_artist_info(artist_id):
    return User().show(artist_id)
```

5.2.1 Image



```
{
    "results": "{'Ludovico_Einaudi': {'song_ids': [987654321, 123456789], 'album_ids': [1], 'playlist_ids': []}}",
    "status": 200
}
```

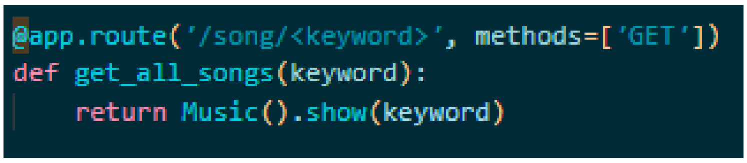
5.2.2 Image

○ Search for Music:

To search for music on the platform, we use a GET endpoint, as shown in the **5.3.1 Image** below. The Python code defines a route (/song/<keyword>) that accepts GET requests and calls the show() method from the Music class. This method retrieves songs associated with the provided keyword.

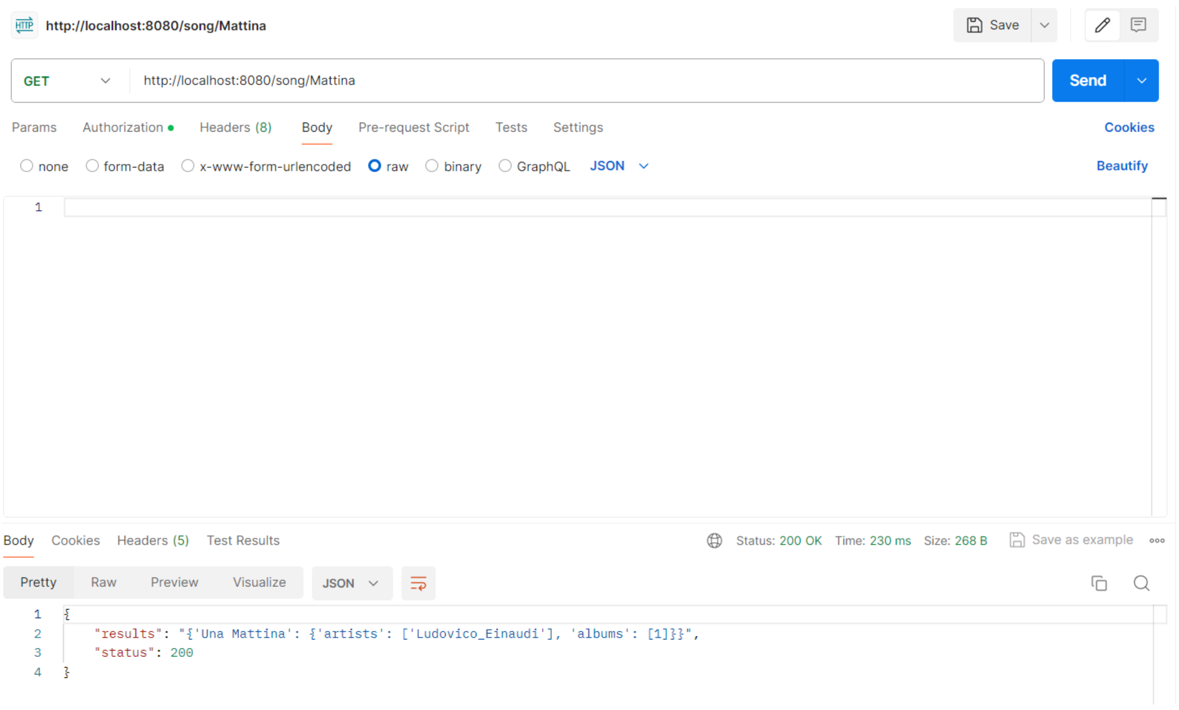
In the **5.3.2 Image**, we see an example of a request being made via Postman, where the keyword used in the URL is "Mattina." The JSON response includes information about the song Una Mattina, associated with the artist Ludovico Einaudi and the album with ID 1. The server responds with a status code of 200, confirming that the data was successfully retrieved.

This process allows for efficient music searches on the platform using keywords, returning detailed information about the found songs.



```python
@app.route('/song/<keyword>', methods=['GET'])
def get_all_songs(keyword):
    return Music().show(keyword)
```

5.3.1 Image



http://localhost:8080/song/Mattina

GET http://localhost:8080/song/Mattina

Status: 200 OK Time: 230 ms Size: 268 B

```
{
    "results": "{'Una Mattina': {'artists': ['Ludovico_Einaudi'], 'albums': [1]}}",
    "status": 200
}
```

5.3.2 Image

○ Make a Subscription:

For a consumer to make a subscription on the platform, we use a POST endpoint, as shown in the **5.4.1 Image** below. The Python code defines a route (/subscription/) that accepts POST requests and calls the subscribe() method from the User class. This method processes the subscription based on the provided data.

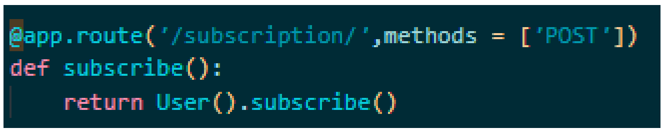
In the **5.4.2 Image**, we see an example of a request being made via Postman. The request body includes the subscription period, which in this case is "quarter," and a list of prepaid card IDs that will be used to pay for the subscription.

There are three plans available on the platform:

- Monthly Plan: costs 7 euros and lasts for 1 month.
- Quarterly Plan: costs 21 euros and lasts for 3 months.
- Semi-Annual Plan: costs 42 euros and lasts for 6 months.

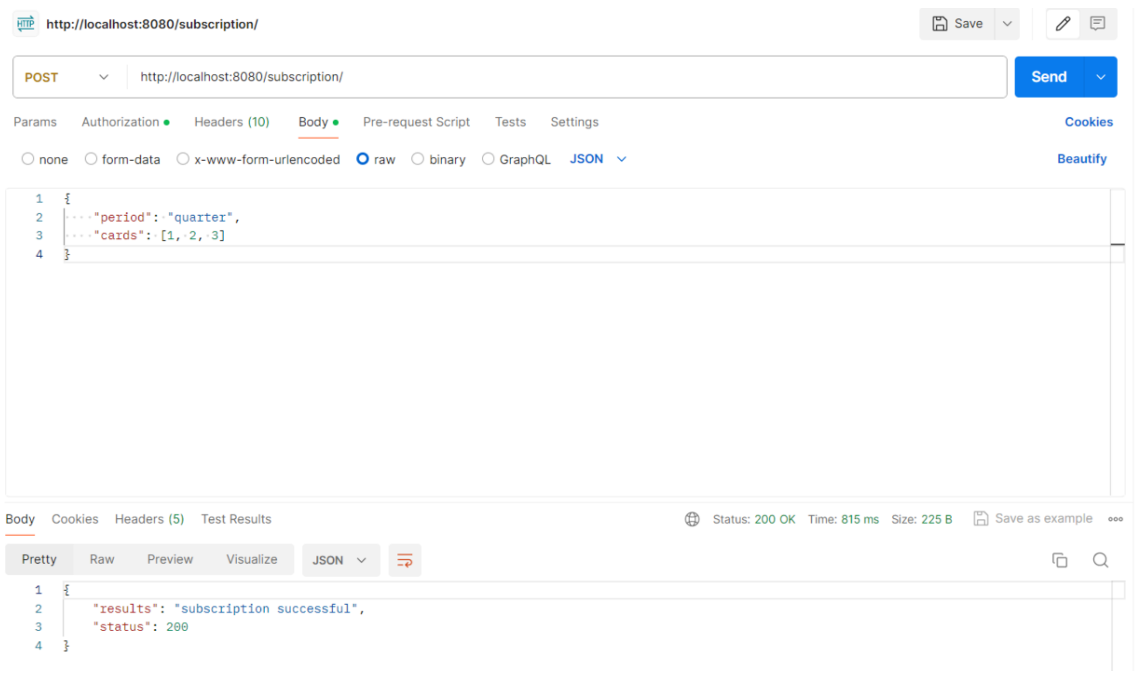
In this example, the values of 3 prepaid cards of 10 coins were combined to cover the cost of the quarterly plan of 21 euros. After submitting the request, the server responds with a status code of 200, confirming that the subscription was successful, as shown in the JSON response.

This process allows the consumer to use multiple cards to complete the subscription payment and enjoy the benefits of the selected premium plan.



```
@app.route('/subscription/',methods = ['POST'])
def subscribe():
    return User().subscribe()
```

5.4.1 Image



http://localhost:8080/subscription/

POST http://localhost:8080/subscription/

```
{
    "period": "quarter",
    "cards": [1, 2, 3]
}
```

Status: 200 OK Time: 815 ms Size: 225 B

```
{
    "results": "subscription successful",
    "status": 200
}
```

5.4.2 Image

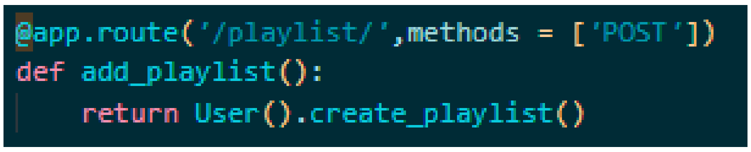
○ Create Playlist:

To create a playlist on the platform, we use a POST endpoint, as shown in the **5.5.1 Image** below. The Python code defines a route (/playlist/) that accepts POST requests and calls the create_playlist() method from the User class. This method processes the creation of the playlist based on the provided data.

In the **5.5.2 Image**, we see an example of a request being made via Postman. The request body includes the name of the playlist and a flag indicating whether the playlist is private or public. The "private" option is an exclusive benefit for platform subscribers. Without a subscription, users can only create public playlists.

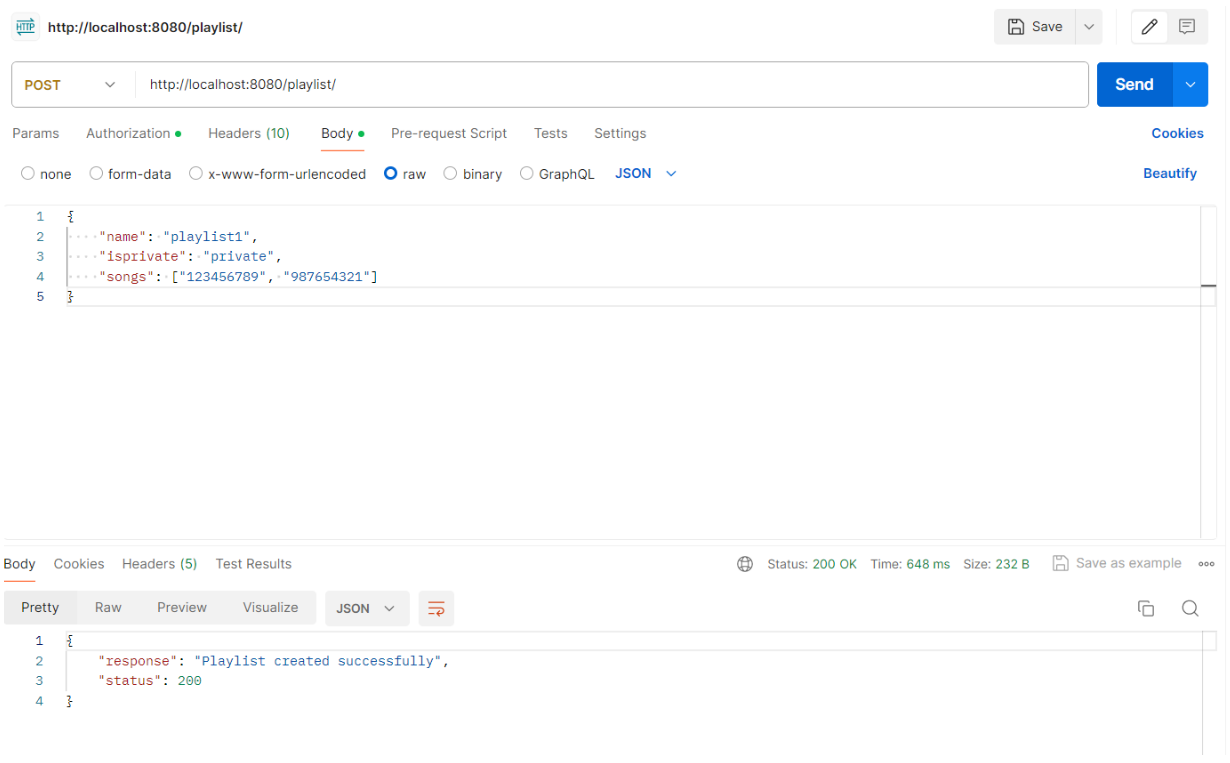
After submitting the request, the server responds with a status code of 200 and a success message confirming that the playlist was created, as shown in the JSON response.

This process allows users to create personalized playlists with their favorite songs on the platform, with subscribers being able to choose between public and private playlists, while non-subscribers can only create public playlists.



```
@app.route('/playlist/',methods = ['POST'])
def add_playlist():
    return User().create_playlist()
```

5.5.1 Image



POST http://localhost:8080/playlist/

```
{
    "name": "playlist1",
    "isprivate": "private",
    "songs": ["123456789", "987654321"]
}
```

Status: 200 OK Time: 648 ms Size: 232 B

```
{
    "response": "Playlist created successfully",
    "status": 200
}
```

5.5.2 Image

○ Make and Reply to Comments:

For a user to comment on a song on the platform, we use a POST endpoint, as shown in the **5.6.1 Image** below. The Python code defines a route (/comments/<song_id>) that accepts POST requests and calls the comment() method from the Music class, processing the comment based on the provided song_id.

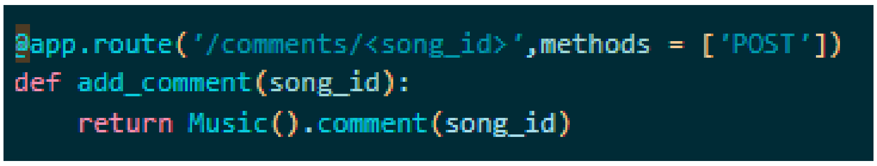
In the **5.6.2 Image**, we see an example of a request being made via Postman. The request body includes the comment "My dream is to go to a Ludovico concert!" and the server responds with a status code of 200, confirming that the comment was successfully added.

To reply to an existing comment, we use a similar route with an additional parameter for the parent comment ID (parent_comment_id), as shown in the **5.6.3 Image**. The Python code defines a route

(/comments/<song_id>/<parent_comment_id>) and calls the reply() method from the Music class to process the reply.

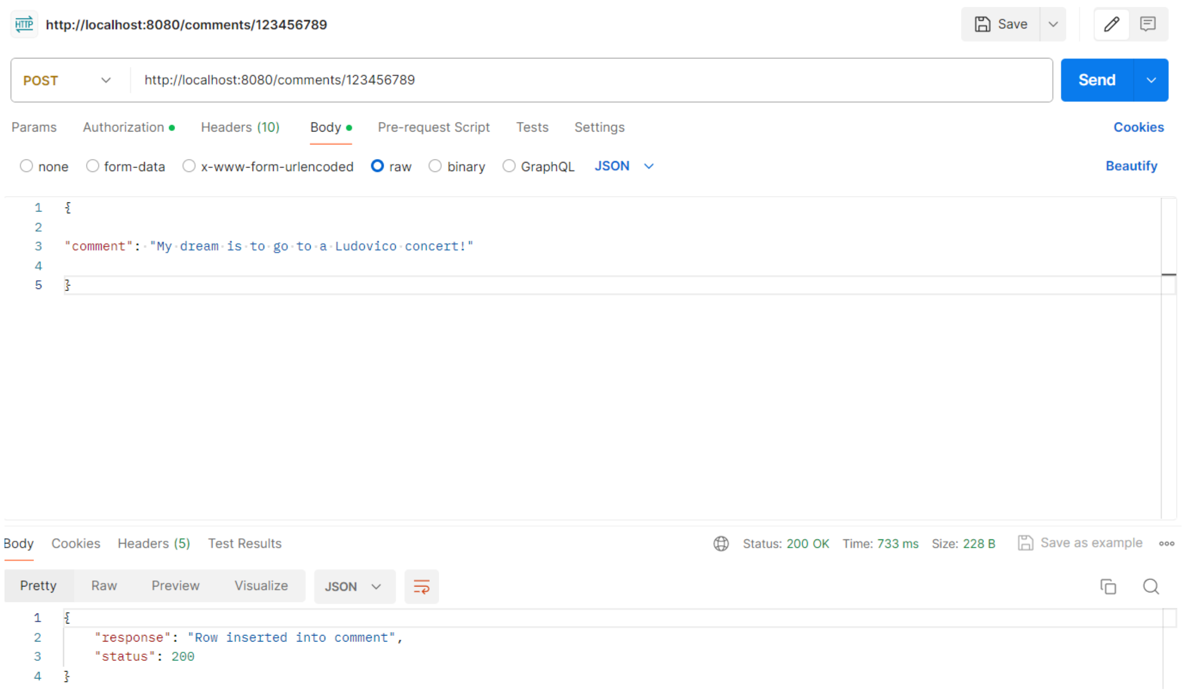
In the **5.6.4 Image**, we see an example of a reply to a comment being made via Postman. The request body includes the reply: "That sounds amazing! Ludovico's music is so captivating live, it must be an unforgettable experience! Hope you get to see him perform soon!" The server responds with a status code of 200, confirming that the reply was successfully added to the original comment.

This process allows users to interact by commenting on songs and replying to comments directly on the platform.



```
@app.route('/comments/<song_id>',methods = ['POST'])
def add_comment(song_id):
    return Music().comment(song_id)
```

5.6.1 Image



http://localhost:8080/comments/123456789

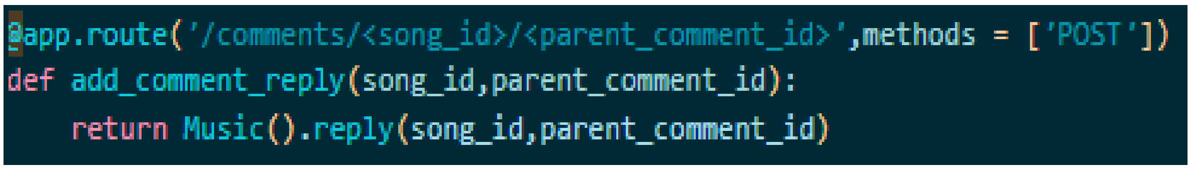
POST http://localhost:8080/comments/123456789

```
{

"comment": "My dream is to go to a Ludovico concert!"

}
```

Status: 200 OK Time: 733 ms Size: 228 B

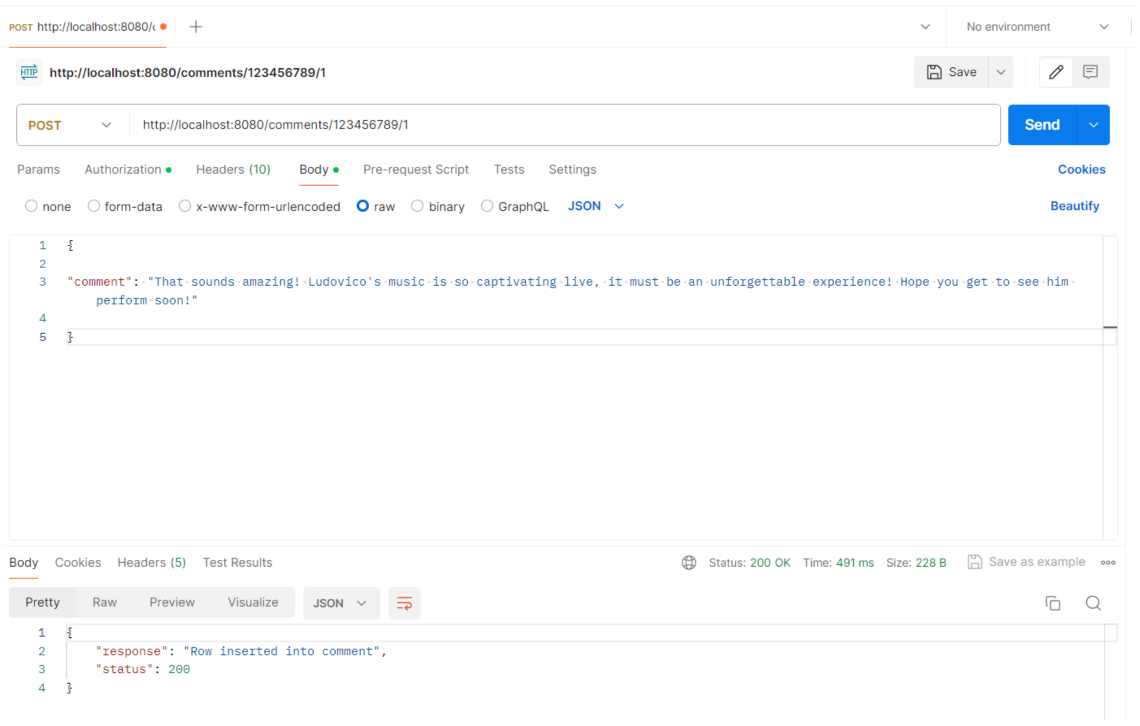
```
{
    "response": "Row inserted into comment",
    "status": 200
}
```

5.6.2 Image



```python
@app.route('/comments/<song_id>/<parent_comment_id>',methods = ['POST'])
def add_comment_reply(song_id,parent_comment_id):
    return Music().reply(song_id,parent_comment_id)
```

5.6.3 Image



POST http://localhost:8080/comments/123456789/1

```
{

"comment": "That sounds amazing! Ludovico's music is so captivating live, it must be an unforgettable experience! Hope you get to see him perform soon!"

}
```

Status: 200 OK Time: 491 ms Size: 228 B

```
{
    "response": "Row inserted into comment",
    "status": 200
}
```

5.6.4 Image

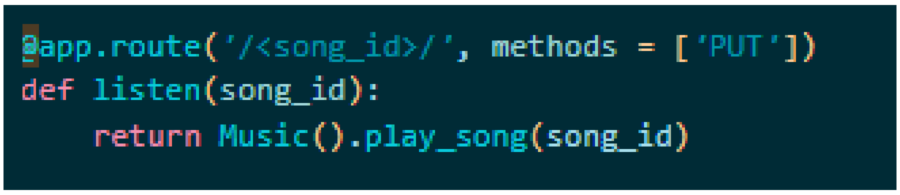
○ Music Playback and Top Playlist:

To register the playback of a song on the platform and update the "top" playlist, we use a PUT endpoint, as shown in the **5.7.1 Image** below. The Python code defines a route (/<song_id>/) that accepts PUT requests and calls the play_song() method from the Music class, processing the playback of the song based on the provided song_id.

In the **5.7.2 Image**, we see an example of a request being made via Postman, where the song with song_id "123456789" was played. The server responds with a status code of 200, confirming that the song was successfully played. The JSON response also shows an update to the top playlist, indicating that the played song now occupies the first position.

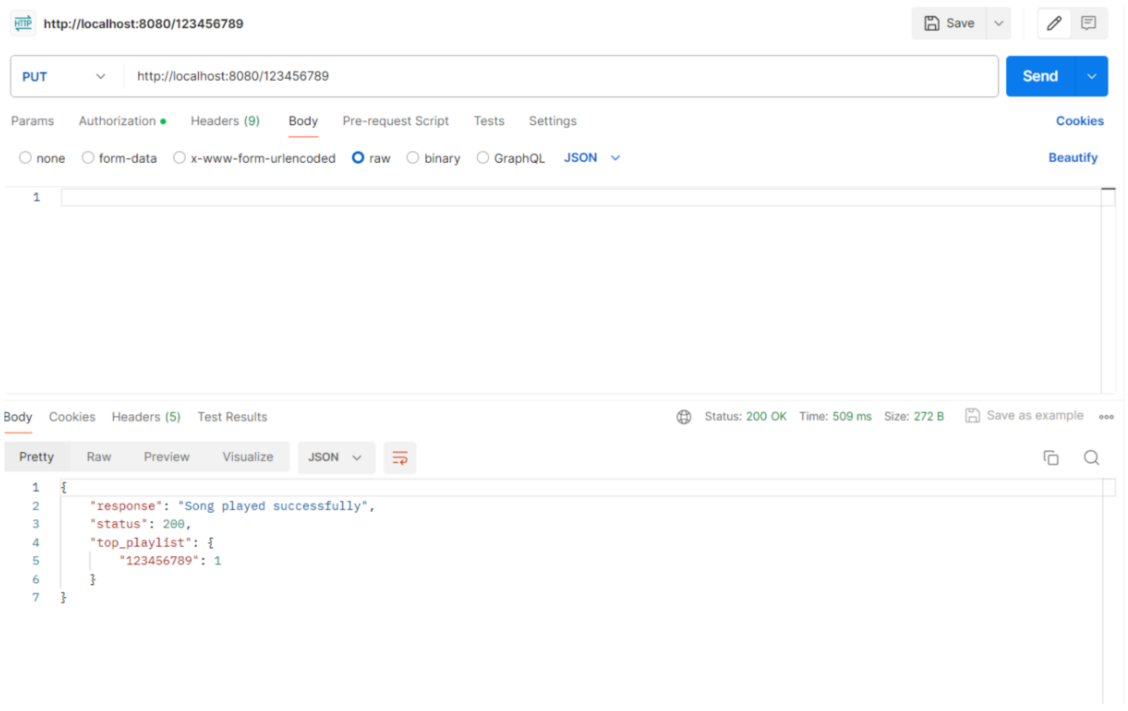
In the **5.7.3 Image**, the song with song_id "987654321" was played. In this scenario, two songs were played: the song "123456789" was played 2 times, and the song "987654321" was played 1 time. This resulted in a shift in the top playlist, with "123456789" maintaining the first position (with 2 plays), followed by "987654321", which appears in the second position with 1 play.

This process allows the platform to track the number of times each song is played and dynamically update the user's "top" playlist based on the most played songs.



```
@app.route('/<song_id>/', methods = ['PUT'])
def listen(song_id):
    return Music().play_song(song_id)
```

5.7.1 Image



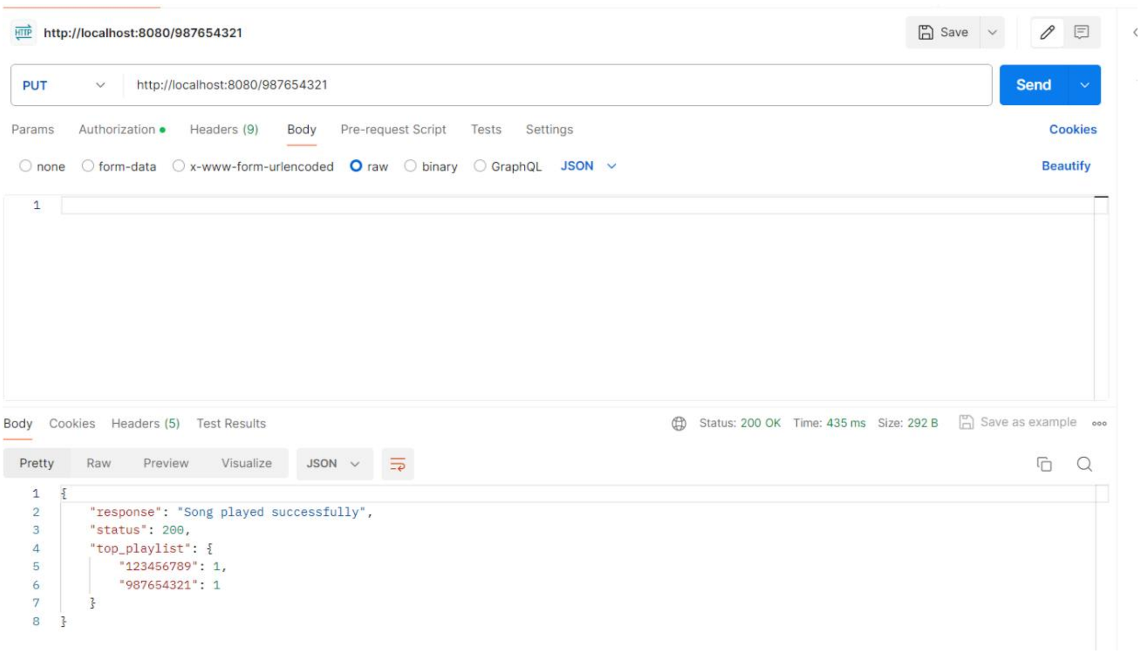
http://localhost:8080/123456789

PUT http://localhost:8080/123456789

Status: 200 OK Time: 509 ms Size: 272 B

```
{
    "response": "Song played successfully",
    "status": 200,
    "top_playlist": {
        "123456789": 1
    }
}
```

5.7.2 Image


http://localhost:8080/987654321

PUT http://localhost:8080/987654321

Status: 200 OK Time: 435 ms Size: 292 B

```
{
    "response": "Song played successfully",
    "status": 200,
    "top_playlist": {
        "123456789": 1,
        "987654321": 1
    }
}
```

5.7.3 Image