UDACITY

< Return to Classroom

Fyyur: Artist Booking Site

REVIEW

CODE REVIEW 5

HISTORY

Meets Specifications

Congratulations, now your project is successfully meeting all the requirements :-)
Besides the specs, I want to highlight the following aspects of your project:

- I liked the fact you gone a step further and used ilike to easily find the search terms.
- Although the UI is quite simple using basically HTML/Javascript, your backend is well structured and could be used with other frontend frameworks.

If you need further technical support focused on your project & lessons, feel free to open a new question on the Knowledge Hub.

Keep up the good work and good luck with the next project!

Data Models

Correct data types are associated with each field.

The `Shows` object has a relationship that connects Artists and Venues, and this relationship is of the correct type. In other words, the project demonstrates the ability to appropriately select from the following types of relationships:

- One-to-one
- One-to-many
- Many-to-many

Good job with your data modeling!

✅ **Shows Relationship:**
A many-to-many relationship fits better for Artists and Venues.

TIPS & REFERENCES
You can also learn other types of relationship using SQLAlchemy.

The code creates a local postgresql database connection.

A local Postgres database was properly configured.

✅ The database URL was provided into config.py.
✅ Migrations scripts were generated using Alembic.

The `Shows` model has properly set up foreign keys.

The Artists and Venues models are in third normal form.

Shows contain the required foreign keys.

✅ Contains Artist reference by its artist_id.
✅ Contains Venue reference by its venue_id.

TIPS & REFERENCES
The secret here is to hold the foreign keys in Shows and handle Artists/Venues models in third normal form, it's a common misconception to do otherwise.

The code uses SQLAlchemy syntax to completely define the models.

The code has accurate SQL queries wrapped in SQLAlchemy commands per API endpoint, calling to define data models and serving expected responses per API endpoint.

The code only uses raw SQL where SQLAlchemy wrappers do not suffice, otherwise minimizing use of raw SQL.

The code is properly using SQLAlchemy without raw SQL statements.

SQL

The code successfully translates SQLAlchemy code for selecting records from the database into the equivalent PostgresSQL command(s) for selecting records from the database.

The code demonstrates correct use of `SELECT` and `WHERE` query statements to execute search successfully.

The search was well implemented using SQLAlchemy and filters.

`JOIN` statements are used to correctly execute joined queries.

The code joins tables from existing models to select Artists by Venues where they previously performed, successfully filling out the Venues page with a “Past Performances” section.

The code joins tables from existing models to successfully fill out the Artists page with a “Venues Performed” section.

he code includes correct equivalents in SQL for all corresponding SQLAlchemy statements.

The `show_venue/show_artist` endpoints are properly handling the past and upcoming shows.

Code connects the New Artist and New Venue forms to a database by successfully using SQLAlchemy to insert new records into the database upon form submission.

Understands the equivalent SQLAlchemy command in SQL syntax, using INSERT INTO.

Code correctly uses SQL constraints to ensure fields that need to be unique, and fields that are required, are given these constraints on the database level, throwing an error if otherwise.

The data validation rules are consistent.

✅ SQLAlchemy was used to insert new records into the database.
✅ Code correctly uses SQL constraints.

Application Quality & Deployment

Code is decoupled into relevant parts across the files.

The code includes good use of comments where there is lack of clarity. Where comments are not provided, the code is self-documenting.

Encapsulate querying code in proper places across Models and API endpoints.

The code is well decoupled and organized!

✅ The code is self-documenting.
✅ Encapsulate querying code in proper places.

- There are no build or compilation errors in running code and launching the web app.
- A user can successfully execute a Search that queries the database.
- A user can view a Venue Page with venue and artist information from the database.
- A user can view an Artist Page with venue and artist information from the database.
- A user can create new venue listing via the New Venue Page.
- A user cannot submit an invalid form submission (e.g. using an invalid State enum, or with required fields missing; missing city, missing name, or missing genre is not required).
- A user can create new artist listings via the New Artist Page.
- A user cannot submit an invalid form submission (e.g. without required fields)
- A user can search for an artist from the venue page, and choose them for a show, specifying a date-time.

Well done, your Fyyur application is working properly!

✅ There are no build or compilation errors in running code and launching the web app.
✅ A user can successfully execute a Search that queries the database.
✅ A user can view a Venue Page with venue and artist information from the database.
✅ A user can view an Artist Page with venue and artist information from the database.
✅ A user can create new venue listing via the New Venue Page.
✅ A user cannot submit an invalid form submission (e.g. using an invalid State enum, or with required fields missing; missing city, missing name, or missing genre is not required).
✅ A user can create new artist listings via the New Artist Page.
✅ A user cannot submit an invalid form submission (e.g. without required fields).
✅ A user can search for an artist from the venue page, and choose them for a show, specifying a date-time.

DOWNLOAD PROJECT

5 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review

START