IE University

Computer Science and Artificial Intelligence

Menu Management System

Designing and Using Databases

2024-2025

David Ezerzer, Gregorio Santi, Simeon Lev, Javier Dominguez

Index

I. PROJECT EXPLANATION

PROJECT OVERVIEW

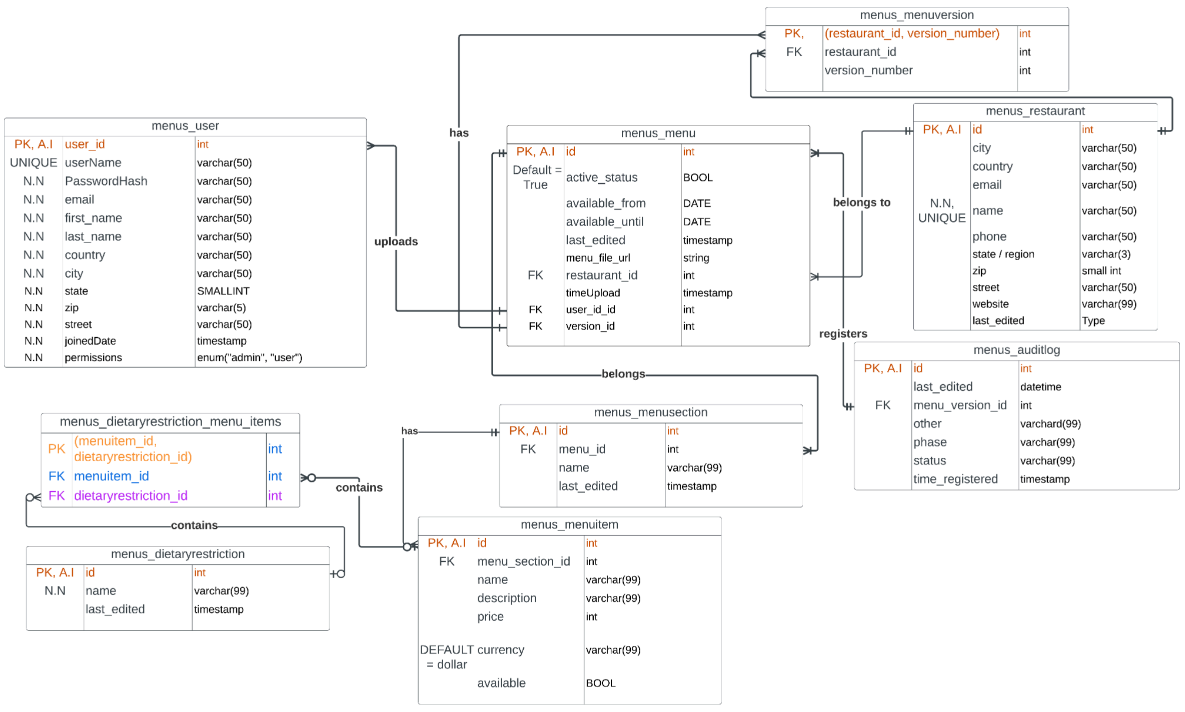
The project involves creating a robust menu management system for restaurants, integrating features such as full-text search, materialized views and database triggers . The system is built using Django for the backend, MySQL for database management, OpenAI APIs for menu processing and AWS for certain cloud infrastructure. Our objective is to stored in a database information about users' uploaded menus by them needing only to upload a file in HTML, PDF or any of the images format (png, jpg, jpeg)

Data integrity is a priority in our project. Even though there is a certain degree of overhead in our ETL pipeline, cleaning and validating data before writing to the database allows us to capture most of the possible edge cases that could potentially disrupt the app's functionality.

Extensive optimization has been done upon the database schema. Indices, full-text search, triggers, events, stored procedures and materialized views have been created for the sake of speeding certain types of queries.

The github of our project may be found here

DESIGN DECISIONS



Our ER diagram. Also accessible here

The database schema is designed in the Third Normal Form (3NF) to eliminate redundancy and ensure data integrity. Each table represents a single entity, and cardinality and relationships are represented through the crow foot notation. We have used an extensive array of data types and constraints in our database schema.

Note the prefix of "menus_" in all the tables; this comes generated automatically from Django, as per the name of the app within the Django project.

Note all non-bridge tables/entities contain a last_edited field. This comes from inspired from the sakila database, we were first encountered this. Keeping track of updates to records provides us with valuable information about the state of the database.

TABLE BY TABLE EXPLANATION:

·**Restaurant**: Contains details like the name, address, and contact information of the restaurant. Acts as a parent table for the Menu entity.

·**Menu**: Stores menu-level information such as the associated restaurant, version, and creation date(time upload) and the link to the S3 blob storage object.

·**Menu section**: Represents individual sections within a menu (e.g., Starters, Desserts), linked to the Menu table.

·**Menu item**: Stores individual items in a menu section, such as name, description, price, and optional dietary restrictions. Currency is default to USD in case there is not one present

·**Dietary Restrictions**: Maintains a list of dietary attributes (e.g., Vegan, Gluten-Free) and is linked to menu item through a bridge table (as many items may have many dietary restrictions and many dietary restrictions may be present in different items).

- **User**: stores information about the user credentials, profile details and priveleges/permissions.
- **Audit log**: for each step of the ETL in the menu a logger is created that will contain information about that phase's state, possible comments and the time it was registered.

-**Menu version:** when a new menu is uploaded if the restaurant name (which is constrained to be unique; may contain location if its a franchise for example) already exists, it gets the latest version, increments it by one and that will the compound key on that table for the give restaurant's menu and the menu's version.

PIPELINE

The ETL pipeline facilitates the seamless integration of new menus into the system's database, regardless of the file format or complexity. It is designed to handle input in HTML, PDF, or image formats, providing a robust solution for extracting, cleaning, and loading data. The process is divided into three key stages: Extraction, Transformation, and Loading.

1. **Extraction**

The pipeline initiates when the user uploads a menu to the django backend. Supported file formats—such as HTML, PDF, and images—are sent to OpenAI's API along with a context message that contains the JSON schema of our database, for which we expect it to be filled.

In the case a menu was uploaded a JSON is returned in markdown. We clean this markdown and extract only the JSON part. If an error occurs when trying to parse the JSON string (that came from the markdown) occurs, an exception is triggered that prints to the stdout the problematic portion of the JSON strings (this was especially meaningful in early testing phases).

2. **Transformation**

The JSON response undergoes a transformation phase, where raw extracted data is processed to ensure consistency, accuracy, and compatibility with the database schema. This step is executed by the extraction.py script, which performs the following operations:

We ensure data integrity by checking for not null columns, transforming currencies symbols into their respective names and reducing the length of varchar() fields that may exceed its allocated space. We also have a robust system of error handling where if the uploaded file is not the content of a menu, an exception will be raised in the ETL process that will exit it prematurely, while reflecting this in the logger system.

3. **Loading**

The final phase involves writing the cleaned and transformed data to the cloud-hosted MySQL database. This step is orchestrated by the insertion.py module, which performs the following tasks:

The insertion.py module ensures that the data is efficiently loaded into the database, preserving accuracy and consistency.

SET-UP

Install all python dependencies with *python3 -r requirements.txt.*
Create a .env file within src/django_project/django_project with the following environment variables:

```
# DATABASE AWS
DB_NAME=
DB_USER=admin
DB_PASSWORD=
DB_HOST=
DB_PORT=

# Django(*)
DJANGO_SECRET_KEY=

# OpenAI
OPENAI_API_KEY=

# AWS S3

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_STORAGE_BUCKET_NAME=
AWS_S3_REGION_NAME=
AWS_S3_CUSTOM_DOMAIN=
```

**Note: you may get the Django secret key running the following thing in your terminal: python -c "from django.core.management.utils import get_random_secret_key; print(get_random_secret_key())"*

Proceed to add to your database the content in the following .sql files:

- dataManipulation.sql

- events.sql
- fullTextsearch.sql
- materializedViews.sql
- triggers.sql

Note indices are added automatically to the database as soon as you make the migrations in your django project (with python3 manage.py makemigrations or running the respective bash script in the /scripts folder)

You may as well need to adjust identified connection to the RDS (AWS) if MySQL > 8.0.0, as per Django having an older version for connecting to it (it uses mysql_native_password):
In your database, run: ALTER USER 'admin'@'%' IDENTIFIED WITH caching_sha2_password BY 'YourCurrentPassword'; FLUSH PRIVILEGES;

II. FEATURES OF OUR PROJECT

QUERIES

5 queries were created to test the results in our database. A single HTTPs endpoint for raw SQL commands execution (not using the ORM). It was defined with a function-based view in views.py. When the django server is running and a request to the endpoint is done, a JSON with the output is returned. Debugging messages (server status code, server response message) may be printed to stdout in the presence of errors.

The queries were saved as stored procedures in the database to easen the process of invocation and for security reasons (all SQL queries expect a format of CALL function(), which assists preventing SQL injections) .

All the further explanation and invocation of the queries may be find here:
QUERIES | DATABASES

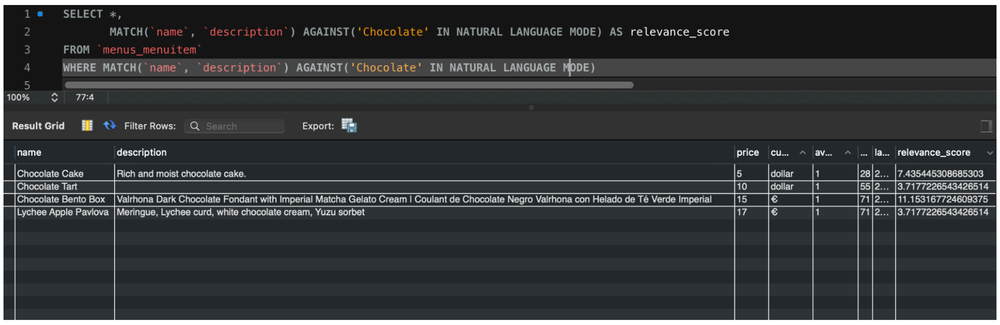
INDICES

Several indices were implemented in order to speed read queries on certain columns. We implemented them through Django's class Meta within models.py. A total of 10 indices were implemented. You may find the justification for all these indices here

FULL-TEXT SEARCH

Full-text search allows to compare a certain table against a certain string (without specifying a column to based it on) and sorts by a priority (the highest is the value where the given string appears the most in the given record's columns)

Its implementation may be find in fullTextsearch.sql



```sql
SELECT *,
       MATCH(`name`, `description`) AGAINST('Chocolate' IN NATURAL LANGUAGE MODE) AS relevance_score
FROM `menus_menuitem`
WHERE MATCH(`name`, `description`) AGAINST('Chocolate' IN NATURAL LANGUAGE MODE)
```

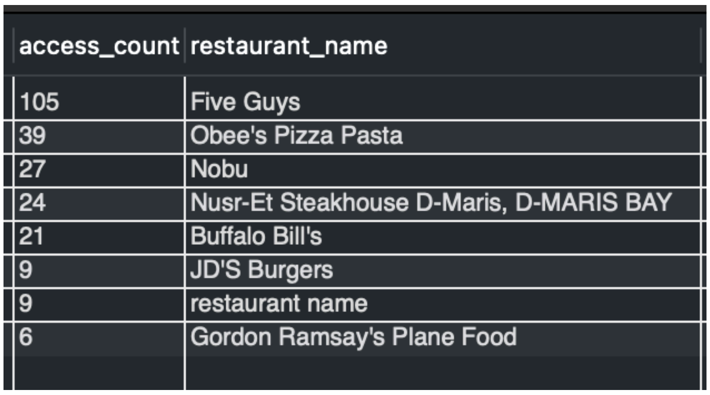
name	description	price	cu...	av...	...	la...	relevance_score
Chocolate Cake	Rich and moist chocolate cake.	5	dollar	1	28	2...	7.435445308685303
Chocolate Tart		10	dollar	1	55	2...	3.7177226543426514
Chocolate Bento Box	Valrhona Dark Chocolate Fondant with Imperial Matcha Gelato Cream I Coulant de Chocolate Negro Valrhona con Helado de Té Verde Imperial	15	€	1	71	2...	11.153167724609375
Lychee Apple Pavlova	Meringue, Lychee curd, white chocolate cream, Yuzu sorbet	17	€	1	71	2...	3.7177226543426514

An example of the relevance score

MATERIALIZED-VIEWS

Materialized views allow to have precomputed tables automatically refreshed on a certain time interval (1 hour in our case). We created a materialized view that provides frequently processed menus. We first had to create a table, then the stored procedure that updates it (truncating it at the beginning and sequentially repopulating), and finally a event that automatically updates the table every hour.

Its implementation may be found in materializedViews.sql.

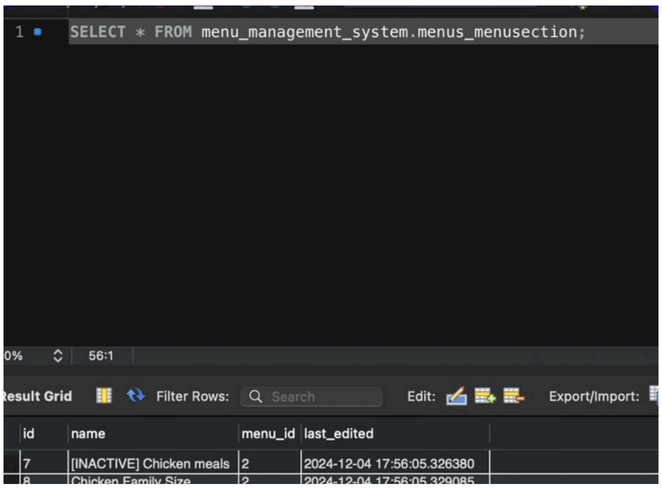


access_count	restaurant_name
105	Five Guys
39	Obee's Pizza Pasta
27	Nobu
24	Nusr-Et Steakhouse D-Maris, D-MARIS BAY
21	Buffalo Bill's
9	JD'S Burgers
9	restaurant name
6	Gordon Ramsay's Plane Food

In our case Five Guys

TRIGGERS

In triggers.sql you may find a trigger whose purpose is to dynamically adjust the name of a section if all its items are disabled.



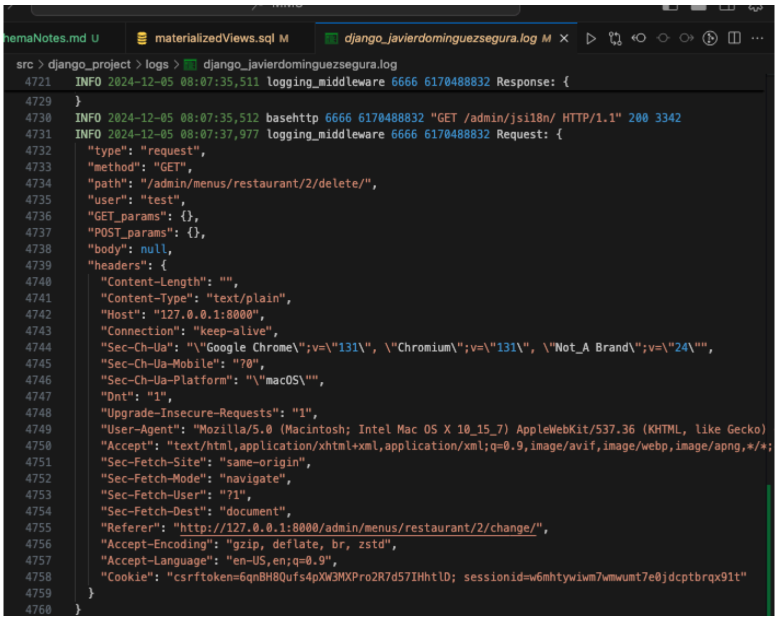
```sql
SELECT * FROM menu_management_system.menus_menusection;
```

id	name	menu_id	last_edited
7	[INACTIVE] Chicken meals	2	2024-12-04 17:56:05.326380
8	Chicken Family Size	2	2024-12-04 17:56:05.329085

An example of the trigger having updated a section after all its items were disabled

LOGGER

An HTTPs-request logger has been established to control all the request that the Django server receives. This is meaningful for debugging and session data storing. Currently is configured to store each user's logger for 30 days.



hemaNotes.md U
materializedViews.sql M
django_javierdominguezsegura.log M
src > django_project > logs > django_javierdominguezsegura.log

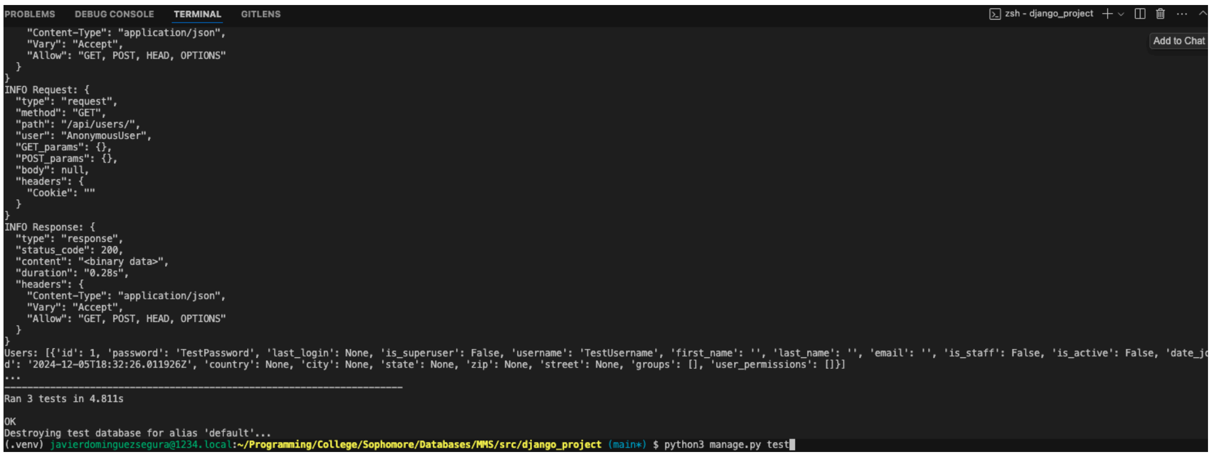
```
INFO 2024-12-05 08:07:35,511 logging_middleware 6666 6170488832 Response: {
}
INFO 2024-12-05 08:07:35,512 basehttp 6666 6170488832 "GET /admin/jsi18n/ HTTP/1.1" 200 3342
INFO 2024-12-05 08:07:37,977 logging_middleware 6666 6170488832 Request: {
  "type": "request",
  "method": "GET",
  "path": "/admin/menus/restaurant/2/delete/",
  "user": "test",
  "GET_params": {},
  "POST_params": {},
  "body": null,
  "headers": {
    "Content-Length": "",
    "Content-Type": "text/plain",
    "Host": "127.0.0.1:8000",
    "Connection": "keep-alive",
    "Sec-Ch-Ua": "\"Google Chrome\";v=\"131\", \"Chromium\";v=\"131\", \"Not_A Brand\";v=\"24\"",
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Ch-Ua-Platform": "\"macOS\"",
    "Dnt": "1",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko)
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;
    "Sec-Fetch-Site": "same-origin",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-User": "?1",
    "Sec-Fetch-Dest": "document",
    "Referer": "http://127.0.0.1:8000/admin/menus/restaurant/2/change/",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "Cookie": "csrftoken=6qnBH8Qufs4pXW3MXPro2R7d57IHhtlD; sessionid=w6mhtywiwm7wmwumt7e0jdcptbrqx91t"
  }
}
```

An example of a captured request from the Django server is stored in the logger

EVENTS

Beyond the event in the materialized views refresher, we have added another event that automatically marks all menus expired if their available data is smaller than present time. This is executed once every day. You may find the code in events.sql

TESTING



```
    "Content-Type": "application/json",
    "Vary": "Accept",
    "Allow": "GET, POST, HEAD, OPTIONS"
  }
}
INFO Request: {
  "type": "request",
  "method": "GET",
  "path": "/api/users/",
  "user": "AnonymousUser",
  "GET_params": {},
  "POST_params": {},
  "body": null,
  "headers": {
    "Cookie": ""
  }
}
INFO Response: {
  "type": "response",
  "status_code": 200,
  "content": "<binary data>",
  "duration": "0.28s",
  "headers": {
    "Content-Type": "application/json",
    "Vary": "Accept",
    "Allow": "GET, POST, HEAD, OPTIONS"
  }
}
Users: [{'id': 1, 'password': 'TestPassword', 'last_login': None, 'is_superuser': False, 'username': 'TestUsername', 'first_name': '', 'last_name': '', 'email': '', 'is_staff': False, 'is_active': False, 'date_jo
d': '2024-12-05T18:32:26.011926Z', 'country': None, 'city': None, 'state': None, 'zip': None, 'street': None, 'groups': [], 'user_permissions': []}]
...
----------------------------------------------------------------------
Ran 3 tests in 4.811s

OK
Destroying test database for alias 'default'...
(.venv) javierdominguezsegura@1234.local:~/Programming/College/Sophomore/Databases/MMS/src/django_project (main*) $ python3 manage.py test
```

See "Ran 3 tests… OK" proving tests were passed

We leveraged Django's testing module to test different components of our application. Specifically, we tested our views, models and serializers to ensure they were working as expected.
The code with the testing can be found in src/django_project/menus/tests.py. You may initialize the testing with python3 manage.py test.
Potentially, these testing modules could be integrated into a CI/CD environment that guarantees that any changes to the codebase may not violate any of these tests.

III. API DOCUMENTATION

Create a superuser using the createuser.bash script, navigate to the scripts folder first. Run the update.bash script to update the database and start the server. Go to http://127.0.0.1:8000/admin/ to access the admin interface. You may be asked for login credentials the first time.

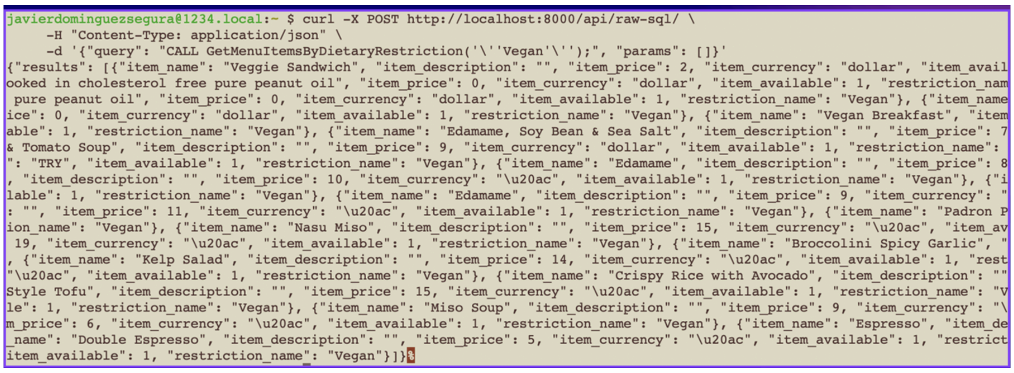
We have nine different tables in our database, or models in the django paradigm: Audit Log, Dietary Restrictions, Dietary Restriction Menu Items, Menu, Menu Item, Menu Section, Menu Version, Users, and Restaurant. Refer to the *ER Diagram* for a detailed explanation of our database's schema. These tables are all in the umbrella of menus which is prefixed to each class in dataDefinitions.sql. All these tables have a respective model in models.py. The tables have a good amount of indexes to speed up querying and ensure relationships we outlined in the ER Diagram. They have been chosen carefully having in mind the indexes' tradeoff between decreasing writing and increasing reading operations.

In the Django app, we have all the functionality behind the Menu Management System. In the django admin front-end, you can create, read, update and delete information for all tables. You are also able to search for records by name, group them by values of a given column. As part of the ETL pipeline when a new menu file is uploaded into the django admin portal, or through a put HTTPs request to the corresponding endpoint, different actions are triggered in the background. These will lead to writing the information of the uploaded menu into the database. Once the menu has been uploaded, extraction.py's functions are called automatically. Here, based on its extension, it will be sent to OpenAI where it is asked to return a JSON that adheres to our database's schema. If no exception occurred during the extraction, insertion.py's functions will be called in order to transform the data such as shrink length if it exceeds the limit and finally, to write to the database in the corresponding table.

In our Django app, our views are split into two parts: function views and class views. Function views are used to have more control over the endpoint's functionality. For other cases, we took advantage of django's flexible class-based views feature that allows for GET, POST, PATCH, PUT, and DELETE requests out of the box. The function views are used for ETL invocation and query execution. For class views, when the assigned url of a specific model class is accessed such as running APIs.py, an HTTPs request is sent out. This request is sent to the specified url in urls.py that will execute the respective code in the assigned view which is in views.py.
For function views, the process is similar to class views, but now we are able to specify in more detail what we want to execute when a request is sent to the HTTPs endpoint.

For our urls to connect to their associated views, our urls.py file is distributed into two parts: class based urls and function based urls. Class based urls register themselves through Django's DefaultRouter which allows each module to connect to a view through its custom url. Function based urls are connected to their respected views through urlpatterns.

As soon as you start the server you may send HTTPs requests to the django backend with the file APIs.py in the folder API. There are classes for all the models, uploading a menu, and sending pure-sql queries. You may instantiate this class and invoke its method with the respective parameters. Please note that for those methods that require a json parameter you just need to send a dictionary with keys being the fields of the table and values being the values of these fields.
A robust exception handling system is provided that will provide informative error messages when the non-successful requests are done.
The APIs.py module provides an easier way to interact with the HTTPs endpoints (as you just need to pass the parameters and the request is done by the function), however you may as well use the curl command in the terminal to get the expected response.



```
javierdominguezsegura@1234.local:~ $ curl -X POST http://localhost:8000/api/raw-sql/ \
     -H "Content-Type: application/json" \
     -d '{"query": "CALL GetMenuItemsByDietaryRestriction('\''Vegan'\'');", "params": []}'
{"results": [{"item_name": "Veggie Sandwich", "item_description": "", "item_price": 2, "item_currency": "dollar", "item_avail
ooked in cholesterol free pure peanut oil", "item_price": 0, "item_currency": "dollar", "item_available": 1, "restriction_nam
 pure peanut oil", "item_price": 0, "item_currency": "dollar", "item_available": 1, "restriction_name": "Vegan"}, {"item_name
ice": 0, "item_currency": "dollar", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Vegan Breakfast", "item
able": 1, "restriction_name": "Vegan"}, {"item_name": "Edamame, Soy Bean & Sea Salt", "item_description": "", "item_price": 7
& Tomato Soup", "item_description": "", "item_price": 9, "item_currency": "dollar", "item_available": 1, "restriction_name":
": "TRY", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Edamame", "item_description": "", "item_price": 8
, "item_description": "", "item_price": 10, "item_currency": "\u20ac", "item_available": 1, "restriction_name": "Vegan"}, {"i
lable": 1, "restriction_name": "Vegan"}, {"item_name": "Edamame", "item_description": "", "item_price": 9, "item_currency": "
: "", "item_price": 11, "item_currency": "\u20ac", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Padron P
ion_name": "Vegan"}, {"item_name": "Nasu Miso", "item_description": "", "item_price": 15, "item_currency": "\u20ac", "item_av
 19, "item_currency": "\u20ac", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Broccolini Spicy Garlic", "
, {"item_name": "Kelp Salad", "item_description": "", "item_price": 14, "item_currency": "\u20ac", "item_available": 1, "rest
"\u20ac", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Crispy Rice with Avocado", "item_description": ""
Style Tofu", "item_description": "", "item_price": 15, "item_currency": "\u20ac", "item_available": 1, "restriction_name": "V
le": 1, "restriction_name": "Vegan"}, {"item_name": "Miso Soup", "item_description": "", "item_price": 9, "item_currency": "\
m_price": 6, "item_currency": "\u20ac", "item_available": 1, "restriction_name": "Vegan"}, {"item_name": "Espresso", "item_de
_name": "Double Espresso", "item_description": "", "item_price": 5, "item_currency": "\u20ac", "item_available": 1, "restrict
item_available": 1, "restriction_name": "Vegan"}]}%
```

Calling a stored procedure in an HTTPs endpoint with curl

IV. CONCLUSION

IMPLEMENTATION CHALLENGES

- Deployment to the cloud:
    - We originally struggled to enable remote access to our database as inbound rules of the RDS were not automatically configured to accept connections from any specific url.
- Selection of columns for indices:
    - Each decision required to have in mind the tradeoffs between faster read and lower writing.
- Design of the database schema:

- We iterated close to 7 times on the original database schema while we were setting up models.py and admin.py. We readjusted the relationships, tables and tables' fields throughout the django implementation of the schema.

AREAS FOR IMPROVEMENT

To enhance functionality and scalability, future improvements could include the integration of real-time updates and notifications to ensure menu changes are instantly reflected across all platforms and communicated to relevant stakeholders. We have also considered expanding the materialized views to provide more insights, such as top-selling items and regional trends.

- Adding support for larger datasets through optimized queries and database sharding could address scalability as the system grows. Database replication could be a useful integration to our system, allowing multiple users to experience consistency in the used database.
- Use DCL (data control language) to restrict privileges based (RBAC- role based access control) on the type of user that has access to the database. You may assign read-only permission for a certain user whose only purpose is to retrieve data (and not necessarily alter it).
- Finally, we would create a more powerful RDS instance. We are currently using t4.micro (for testing purposes) which has limited processing-power.
- Providing a front-end for a user-friendly interaction with our application