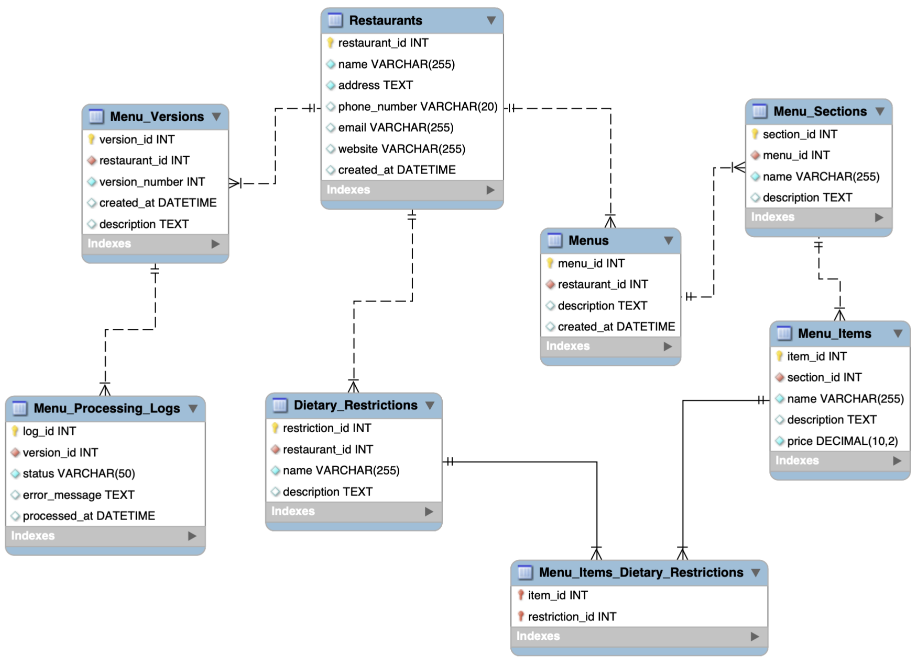
Documentation
IE University
Jose Antonio Garcia

By: Boris Hendrik, Gabriela Vega, Karem Rami and Fares Shawkat

Project Overview:

Our project's objectives are to manage restaurant data using a Django-based web application, automate the extraction of menu data from PDF files, and transform it into structured JSON format. The main features include the ability to upload PDF files, process and parse them using AI-based models and optical character recognition (OCR), and store the information extracted in a database. Users can also access and manage restaurant menu data with the application.

ERD:



Key Entities and Relationships:

Entities:

1. Restaurants: Shows the individual restaurants in the system. It shows the basic information about a restaurant, containing details such as name, address, website and more.
2. Menus: Shows the menu details in each restaurant
3. Menu Sections: Shows the items in sections (Main Course, Desserts, etc)

4. Menu Items: Shows menu item information (Price, name of item)
5. Dietary Restrictions: Represents dietary restrictions for items (Gluten Free, Vegan, etc)
6. Menu Items - Dietary Restrictions: Represents a bridge table that connects menu items to dietary restrictions
7. Menu Versions - It tracks different versions of a menu for each restaurant
8. Processing Logs: Tracks processing status of menu PDFs

Relationships:

1. Restaurants to Menus: A restaurant can have many menus (1:N relationship). (one to many)
2. Menus to Menu Sections: Each menu contains multiple sections (1:N relationship). (one to many)
3. Menu Sections to Menu Items: Each Menu section contains multiple menu items (1:N relationship). (one to many)
4. Menu to Menu Versions: Each menu can have multiple versions (1:N)
5. Menu Items to Dietary Restrictions: Multiple menu items can have multiple dietary restrictions (M:N relationship when using the bridge table). (many to many)
6. Menus to Menu Processing Logs: Each menu has an entry log (1:N relationship). (one to many)

Index Strategy and Constraints

Each table has its own primary key that creates an index. This index is essential as it ensures a fast and efficient query. Foreign keys help maintain data integrity and establish relationships between tables.

Primary keys

Table Name	**Primary Key**
Restaurants	restaurant_id
Menus	menu_id
Menu Sections	section_id
Menu Items	item_id
Menu Versions	version_id
Dietary Restrictions	restriction_id
Menu Items Dietary Restrictions	Composite: item_id, restriction_id
Menu Processing Logs	log_id

Foreign Keys

Table Name	**Foreign Key(s)**
Menus	restaurant_id
Menu Sections	menu_id
Menu Items	section_id
Menu versions	restaurants_id
Dietary Restrictions	restaurant_id
Menu items Dietary Restrictions	Composite: item_id, restriction_id
Menu Processing Logs	menu_id

Unique Constraints

They ensure that no duplicate values are in the key fields:
- Restaurants.email must be unique (no two restaurants can share an email)
- Menu versions - menu_id and version_number

Not Null Constraints

These constraints ensure that certain fields must always have a value, making sure key data points are always provided. In total, we have 16 fields in total that are not null (Can be seen in the schema charts below).

Default values

Default values handle situations where a value is not provided. This is useful when you want to automatically populate a field with a predefined value or current timestamp. (Can be seen in our schema charts below)

Indexing Strategy

Index-Type	**Key Elements**	**Purpose**
Page Based	Numeric page indices, error tracking per page	For modular isolation
Section Based	Menu data is divided into sections, each with an indexed section_name	For a logical organization of the menu contents

Error Based	Errors are indexed in the errorMessages dictionary	Detailed logging for debugging purposes
Json Object	Nesting indexing of sections and menu items	Hierarchical organization of the extracted data
Page to Image	Mapping pdf pages to images with base64 encoding	Accurate association of image data with pages

How we follow ACID Properties

1. **Atomicity**:
    - Ensures that the transactions either all succeed together as a whole or fail. This is to prevent things from being incomplete with insufficient data.
    - For example, in our database, the nut null constraints show that either the data is all there and accepted or rejected if it's incomplete. We can also see that when adding the associated menu items to a new restaurant, the transaction reverts if it fails (inserting into restaurants and menu_items).
2. **Consistency:**
    - This ensures that the data is consistently valid and maintained with the use of rules like foreign keys and constraints, it prevents corrupt data from being saved.
    - For example, the primary keys ensure that every entry is consistently valid.
3. **Isolation**:
    - This prevents concurrent transactions from clashing with each other, it's also very useful in preventing any clashes when multiple users connect at the same time. With the use of MySQL in our database, isolation levels such as READ COMMITTED and SERIALIZABLE handle simultaneous access to ensure that there is no corruption.
4. **Durability**:
    - Ensures the backup of any transaction that is complete, transactions are permanently saved in the database so that if the system crashes they are still intact. In our database, once a restaurant or a menu item is added, MySQL will save its data with the use of its backup and automatic logging features.

Schema

1. **Restaurants:**

# Field	Type	Null	Key	Default	Extra
restaurant_id	int	NO	PRI	NULL	auto_increment
name	varchar(255)	NO		NULL	
address	text	NO		NULL	
phone_number	varchar(20)	YES		NULL	
email	varchar(255)	YES	UNI	NULL	
website	varchar(255)	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

2. Menus

# Field	Type	Null	Key	Default	Extra
menu_id	int	NO	PRI	NULL	auto_increment
restaurant_id	int	NO	MUL	NULL	
description	text	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

3. Menu Items

# Field	Type	Null	Key	Default	Extra
item_id	int	NO	PRI	NULL	auto_increment
section_id	int	NO	MUL	NULL	
name	varchar(255)	NO		NULL	
description	text	YES		NULL	
price	decimal(10,2)	NO		NULL	

4. Menu Versions

# Field	Type	Null	Key	Default	Extra
version_id	int	YES	PRI	NULL	auto_increment
restaurant_id	int	NO	MUL	NULL	
version_number	int	NO		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	
description	text	YES		NULL	

5. **Menu Sections:**

# Field	Type	Null	Key	Default	Extra
section_id	int	NO	PRI	NULL	auto_increment
menu_id	int	NO	MUL	NULL	
name	varchar(255)	NO		NULL	
description	text	YES		NULL	

6. **Dietary Restrictions**

# Field	Type	Null	Key	Default	Extra
restriction_id	int	NO	PRI		auto_increment
restaurant_id	int	NO	MUL		
name	varchar(255)	NO			
description	text	YES			

7. **Menu Items Dietary Restrictions**

# Field	Type	Null	Key	Default	Extra
item_id	int	NO	PRI		
restriction_id	int	NO	PRI		

8. **Menu Processing Logs**

# Field	Type	Null	Key	Default	Extra
log_id	int	NO	PRI		auto_increment
menu_id	int	NO	MUL		
status	varchar(50)	NO			
error_message	text	YES			
processed_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Design Justification:

Normalization:

Eliminates data redundancy, the database corresponds to the Third Normal Form (3NF) for multiple reasons:

- Each table in the database stores data about a single relationship or entity, eliminating data redundancy. For example, the Restaurant table stores details about restaurants. Dividing the data into separate tables with the matched foreign key eliminates data redundancy.
- Each non-key attribute in a table depends on the primary key of that table, eliminating partial dependencies in tables with composite primary keys. For example, In the Menu Items table, columns like name, price, and description depend only on menu_item_id.
- Transitive dependencies are removed, ensuring that every attribute is directly related to the primary key of its table. For example, dietary restrictions for a menu item are stored in the Dietary Restrictions table, defining them using foreign keys rather than duplicating the data in the menu items table.

Querying

The design makes querying easier since it supports straightforward joins between tables. For example, it executes queries such as retrieving all items in a menu or using a JOIN query between the Menus and Menu Sections tables on their shared key, menu_id. This allows us to easily find all of the sections for this particular menu (menu_id), and by using another JOIN query with the menu_items table, we can easily find every menu_item with the corresponding section_id and, thus, menu_id. Apart from simplifying querying by removing the need for complex conditions, by changing this method, we can also find every menu item within a specific section, giving us flexibility in how we display a menu.

Most Important Queries

Upload_full_menu

First Transaction

- Uses transaction.atomic() to ensure that the initial code block executes reliably
- Only depends on required inputs validated by the form and JavaScript
- Log is always created regardless of API success

Second Transaction

- Encapsulates processing all of the other data included in each section
- Each section uses transaction.savepoint(), isolating its execution: savepoint_commit() is called for success and savepoint_rollback() is called for recoverable errors. This Ensures that a failure in one section doesn't affect others
- Within each section there exists at least one menu_item. For each menu_item it checks if it has a dietary_restriction. If it does, it initialises another transaction.savepoint() that again isolates the dietary restriction uploads from everything else

Get_menu_data

- It queries the database to find all the associates information for a given menu_id
- It first queries for the specified menu object with .prefetch_related that references all related objects that we care about
- We first initialise the actual menu object by using menu.first() and then initialize the version object as it has a many to one relationship
- Then loop through each section which is defined by menu_data.sections.all() and then it does the same do the same for each menu_item defined by section.items.all()
- For each section we use an aggregate function to get the average price per section. We display these and also use them to get the average menu price while minimizing queries

Get_all_restaurant_data

- It fetches data for all restaurants to support an auto-fill feature for menu uploads
- The value for each restaurant includes its address, phone number, email, and website
- By using 'Restaurants.objects.values(...) instead of just 'Restaurants.objects.all()' it makes sure that the query is only getting what it needs from the table
- Although in our case this means that only two columns are skipped in the table (id and created at), it's good practice and also improves the scalability of the app in case we wanted to add more restaurant information in the future

Method	**Purpose**	**Results/Outputs**
upload_full_menu	It uploads a menu complete with sections, items, and dietary restrictions	It saves data to the database and logs processing status.
get_menu_data	It retrieves information about a specific menu	It will output a JSON-like dictionary with menu, section, and item details
get_all_restaurant_data	Used for the auto-fill capability when uploading a menu. It receives all of the relevant	Outputs a dictionary of restaurant names and their details

	information about a restaurant and sends it to the frontend	

Challenges

The biggest challenge throughout this project was figuring out where to start. We were unsure of how to start parsing PDFs as they are notoriously unstructured. After experimenting with multiple libraries, including Tesseract (OCR), our initial strategy involved identifying sections by examining text attributes (font size, boldness, color, etc.) . This approach was too complicated, though, so we decided to instead submit photos straight to the API, giving up speed for accuracy in exchange for logs and real-time updates. Another really big challenge was optimizing Django's database queries. Initially, performance was hampered by ineffective "N+1" queries. This was fixed by moving to prefetching and refining models with stronger associations (like related_name), although it necessitated rewriting queries. Lastly, managing error messages was very challenging since including them in API responses complicated efficient queries due to unpredictable input. Ultimately, we reworked the API request with a more precise prompt which simplified the upload_full_menu process.

Future Improvements

Some future enhancements that could be made are adding login functionality to make the experience more personalized for the user. Besides this, enhancing the editing page by preserving the original menu layout (background, font, etc.) and overlaying it on the parsed menu would improve the UI and help users spot parsing errors more efficiently.