**Project Overview: The Localized Outdoor Recreation Database (LORD)**

* **Original Objective:** Create a localized dataset for Arizona, Oregon, and Utah, providing a leaner, more agile database accessible for locating hiking, camping, and other outdoor recreation activities in our home states. The Recreation.gov API was selected as the data source: https://ridb.recreation.gov/
* **Implementation**: The project has now met the original objective and is pulling only the data for the selected states with camping activities.
* **Initial Data Model:** We started with a basic understanding of the data structure, using a mix of Python data types (e.g., bool, float64, int64, and object), which we later mapped to PostgreSQL types.
* **Table Creation:** We used CREATE TABLE statements within pgAdmin and also used scripts, to define the schema, with an initial set of columns such as:
  + Facilities (general facility information)
  + Campsites (details about campsites within facilities)
  + Activities (available activities at each facility)
  + PermittedEquipment (Permitted equipment at each campsite)
  + FacilityAddress (addresses associated with each facility)
  + CampsiteAttributes (Additional Attributes for the campsite)
* **Data Loading:** We used the COPY command to load CSV data into the corresponding tables.
* **Primary and Foreign Keys:** We used ALTER TABLE to set up the primary key constraints and the foreign key relationships to ensure data integrity and to link the tables based on those key values.

**II. Our Data Engineering Journey: The Scripting Phase**

* **Initial Steps (Jupyter Notebook):** We began by creating a Jupyter notebook to explore the available data quickly. This allowed us to test our logic and review the data quickly. We were able to quickly make API calls, see data frames, and understand the structure of the API responses.
* **Transition to Python Scripts:** We moved to a more production-ready method, and therefore, we created separate Python scripts for the following:
  + fetch\_and\_save\_data.py: Responsible for fetching all of the data from the API (main facilities, activities, and campsites data) and saving it to a single local JSON file.
  + create\_dataframes.py: This file is used to pull the local JSON file, process the data to create multiple data frames, and output them to a series of CSV files.
* **API Key Management:** We removed hardcoded API keys from our code by moving it to a config file and adding that file to .gitignore.
* **Generalizing the fetch method:** We added flexibility to our fetch script by allowing us to pull from different API endpoints, by using a new endpoint parameter, also added a states parameter to pull data from various states.
* **Rate Limiting and Timeouts:** We used time.sleep() to ensure we were respecting rate limits of the API, and we also added timeouts to improve reliability when API's take too long to response. We also made changes to our fetch logic to adhere to the different rate limits for main calls vs related calls.
* **Data Normalization and Duplication Issues:** We have removed redundant nested data by creating separate dataframes.
* **Improved Output Feedback:** We added time stamps to the progress output so you can know when new API calls are being made, as well as how long those calls are taking.
* **Incomplete Data:** We fixed an issue where the Address data was missing by including full=true in our fetch statement.
* **Data Types and Structure** We also added checks and constraints for various data types to ensure that we were outputting consistent data.
* **Incorrect Looping:** We corrected an error that caused some of our loops to terminate too early and not pull all of the data.
* **Unnecessary Code:** We have removed code that was not needed, as well as functions that were no longer being used. We also removed API calls from the data processing file, so it was more focused on data transformation, improving readability and performance.
* **Missing Imports**: We added import statements where they were missing to resolve run time errors.
* **Incorrect Data Checks** We refined the data checks to properly load all of the data and prevent exiting loops too early.
* **Incorrect Column Names:** We identified and corrected several column name issues.
* **JSON String Format:** We identified that the GEOJSON data included single quotes, which made it not a valid JSON format. We corrected this by replacing the single quotes with double quotes.
* **Over-Inflated Table** We identified that we were creating duplicate data when we merged in the permitted equipment information and then fixed that by not including the joined tables in our final version.

**III. Database Design and Loading:**

* **Initial Data Model:** We started with a basic understanding of the data structure and iteratively refined it into a more normalized schema. Then, we used a mix of Python data types (e.g., bool, float64, int64, and object), which we later mapped to PostgreSQL types.
* **Table Creation:** We used CREATE TABLE statements within pgAdmin and also used scripts to define the schema based on our current dataframes.
* **Data Loading:** We used the COPY command to load CSV data into the corresponding tables.
* **Primary and Foreign Keys:** We used ALTER TABLE to set up the primary key constraints and the foreign key relationships to ensure data integrity and to link the tables based on those key values.
* **Type Mapping:** We adjusted the column types from our starting set to TEXT (for strings), FLOAT (for decimals), BIGINT (for large integers), and BOOLEAN for boolean values in PostgreSQL.
* **Missing data:** We addressed issues with NULL values by modifying the schemas to remove the NOT NULL constraint from columns that can have NULL values.
* **Duplicate Key Errors:** We encountered duplicate key errors when loading data and used TRUNCATE TABLE and ALTER TABLE statements to resolve them, and used the ON CONFLICT DO UPDATE and ON CONFLICT DO NOTHING when appropriate to skip or update records based on duplicated values.
* **CSV Formatting Issues:** We addressed unterminated CSV quoted field errors using tools to debug and fix the CSV structure, ensuring correct data loading and removing the unnecessary ESCAPE attribute from your COPY statement.
* **Column Type Issues:** We addressed the invalid input syntax for type integer errors by changing your column type to TEXT and by correcting the types in your CSV data.
* **Incorrect Data Format:** We used SQL queries to identify malformed data, and to filter out those records from your results.
* **Validating Records:** We added SQL statements to filter your records and check that your data has the proper format, and that specific values are as you expect them to be.
* **JSON Formatting**: We fixed a formatting error where JSON string values in the GEOJSON field were using single quotes, and used replace() to change the format to double quotes which is required by most JSON parsers.

**IV. Remote Access: Building the FastAPI API:**

* **FastAPI as the Backbone:** We used FastAPI as the core framework for creating our API.
* **psycopg2 Integration:** We utilized psycopg2 to connect to the PostgreSQL database and execute SQL queries.
* **Initial Endpoints:**
  + /facilities: To fetch a list of facilities with optional filters for state and ADA accessibility.
  + /campsites: To retrieve campsite information, including AttributeName and MaxLength, along with the option to filter the values based on a state.
  + /activities: To retrieve a list of activities, including the facility and activity name, and a description, with the option to filter the value based on a state.
* **JSON Responses:** All the API endpoints returned data in JSON format, to allow for it to be easily parsed.
* **Adding a Welcome Page:** We created a root endpoint /, that displayed the name of your API, as well as a list of all the valid API endpoints.
* **Authentication:** We used environment variables and a config.py file to store and manage the database password, so it is no longer hardcoded in your code.
* **Extending the API:** We added more API endpoints for accessing data from each of your tables using the /all\_tablename naming convention.
* **Filtering and Error Handling:** We added filters by state and ADA accessible, and we handled JSON decode errors by adding a try..except block, and by sanitizing the data using regular expressions, and skipping over records with invalid formatting.
* **API Descriptions:** We improved the display of the welcome page by displaying the HTTP method and adding a custom summary to each endpoint. We also set up a description for the /redoc endpoint.

**V. The Dashboard (Streamlit with Leaflet):**

* **Streamlit as the User Interface:** Streamlit was the framework of choice to construct a simple, interactive web-based dashboard.
* **Initial Setup:** We started by creating an initial dashboard with streamlit\_folium to display a map and to filter data based on the parameters specified in the sidebar, and by using the data from the /facilities endpoint.
* **Map and Filtering:**
  + The dashboard displayed a map of facilities using Leaflet, using either a GEOJSON object or latitude and longitude values to render each point.
  + The dashboard included a sidebar for filtering by state and also to filter by ADA accessibility.
  + The code was adjusted to use explicit filtering of the data using SQL queries to ensure data integrity.
* **Display of all records:** We added API calls and logging to be able to inspect and debug our code. We added log output for the API calls, the response, and if there were any errors during the render phase, and also to make sure that when values could not be rendered to a map, we would log it in the screen as well.
* **Error Handling:** We handled exceptions from failed API calls and displayed those errors on screen to allow you to debug the problems.
* **Styling and Display:** We added a more descriptive rendering in the home page, to show a list of all your API endpoints, including a description, a method, and a full path to the endpoint. We made several adjustments to the styling to ensure that the output is what you expect to see.
* **Removed console log output:** We removed all the code that was adding console log outputs, to create a clean UI.
* **Removed group\_site filter:** We removed the group\_site parameter from the API and from the dashboard.
* **Used explicit filtering:** We improved the code to make sure that we filter the data based on the latitude and longitude, as well as the JSON format using SQL queries. We added extra checks in the API code to prevent the API from crashing.
* **Initial Display:** The code was simplified to only show the map, and no other data components, and the map was initially hardcoded, and it was only updated when we had verified that everything else was working correctly.
* **Removed Javascript implementation:** We moved away from using a Javascript implementation with D3 and Leaflet and decided to use the streamlit\_folium library since it was easier to use and simpler to debug.
* **Refined Data Handling:** We improved the data handling to ensure that all records were processed and displayed correctly in the map, and to verify that all parameters and data types were consistent and used correctly in the code.

**VI. Key Libraries Used:**

* fastapi: For building the RESTful API.
* uvicorn: An ASGI server for running the FastAPI application.
* psycopg2: For connecting and interacting with the PostgreSQL database.
* streamlit: For the rapid creation of the web-based dashboard.
* requests: For making API calls from the Streamlit dashboard and for the fetch and save data script.
* folium: To generate interactive maps.
* streamlit-folium: To show interactive maps in Streamlit.
* python-dotenv: To load your DB\_PASSWORD from environment variables.
* json: For working with JSON data.
* re: To perform regex operations, such as adding double quotes to keys, when your JSON values are not properly formatted.
* os To handle operating system interaction such as file paths, folder creation etc.
* datetime To handle time-based logic such as tracking time elapsed, or time stamping logs.

**VII. Key Takeaways and Next Steps**

* **Database Schema Evolution:** We started with a basic understanding and progressively refined it based on the data and requirements to be more accurate to the structure of the data. We created a proper SQL database with many different tables and relationships.
* **Error handling**: The code has a wide variety of built-in checks and error handling to catch many kinds of problems that may occur.
* **Incremental Approach:** We moved incrementally, adding functionality in small chunks, which allowed us to identify problems quickly and fix them efficiently.
* **Collaboration:** By sharing our progress and by communicating using specific error messages, we were able to quickly identify issues and fix them efficiently.
* **Clean Code:** By following the principle of Separation of Concerns we made sure to make code as manageable and organized as we could.
* **Flexibility** We implemented flexibility throughout the code, so that the code could be changed and updated based on new requirements or conditions.
* **Data Normalization:** By reviewing the structure of the data, we made sure that it is properly normalized by creating separate tables for each entity, and connecting those entities using foreign keys.
* **End-to-End Pipeline:** You have developed a complete, end-to-end ETL pipeline (from data extraction to database loading to UI design) to address your initial project goals, as well as create an API for access.
* **API Calls**: You have a very strong understanding of when the API is being called, and have updated the code to minimize the number of calls you make