Task 4: Data Loading and Main Dashboard

Objective: To load the exoplanet data from the CSV file using Pandas and display it in a dynamic, sortable, and filterable table on a protected main dashboard page.
Estimated Time: 3-5 hours

Step 1: Create the Data Loading Service

File to Modify: app/analysis/core.py
Action:
This is the first time we're touching the dedicated analysis module. Its job is to handle all data logic, completely separate from the web routes.

1. Import the pandas library. You may need to add it to your requirements.txt and pip install it if you haven't already.
2. Create a function, let's call it load_exoplanet_data(filepath).
3. Inside this function, use pandas.read_csv(filepath) to load your exoplanet_data.csv file into a DataFrame.
4. **Perform basic data cleaning.** Real-world data is messy. For now, let's focus on a few key columns that might be useful for display (e.g., pl_name, hostname, discoverymethod, disc_year, pl_orbper - orbital period, pl_masse - planet mass in Earth masses). Use the .dropna() method on the DataFrame to remove rows that have missing values in these essential columns.
5. The function should return the cleaned Pandas DataFrame.

Why: Separating data logic into its own file (core.py) is a crucial design pattern. It means your data analysis code is not tied to Flask. You could import and reuse this load_exoplanet_data function in a Jupyter Notebook, a different script, or a future API without changing a single line.
Documentation:

- Pandas read_csv
- Pandas dropna

Step 2: Build the Main Dashboard Route

File to Modify: app/main/routes.py
Action:

1. Import login_required from flask_login, your load_exoplanet_data function from app.analysis.core, and the current_app object from flask.
2. Create a new route for /dashboard. Make sure to protect it with the @login_required decorator.
3. Inside the dashboard route function, construct the full path to your CSV file. You can use current_app.root_path to help build a reliable path to the data/exoplanet_data.csv file.
4. Call your load_exoplanet_data() function to get the DataFrame.
5. Pandas DataFrames have a handy .to_html() method. However, for more control, it's

better to convert the DataFrame to a list of dictionaries using .to_dict(orient='records'). This gives you a structure that's easy to loop through in your Jinja2 template.
6. Pass the list of planet data to your template using render_template.

Why: The route's job is to be the "controller." It fetches data from the analysis "service" and passes it to the "view" (the template). It doesn't know *how* the data was loaded, only that it needs to get it and display it.

Documentation:

- Pandas to_dict

Step 3: Create the Dashboard Template

New File: app/templates/main/dashboard.html

Action:

1. Make this template extend your base.html.
2. Create a standard HTML <table> with a <thead> for your column headers (Planet Name, Host Star, Discovery Year, etc.) and a <tbody>.
3. Use a Jinja2 for loop to iterate over the list of planet dictionaries you passed from the route.
4. Inside the loop, create a table row (<tr>) for each planet and table data cells (<td>) for each piece of information.

Why: This step renders the raw data into a structured format. At this point, you'll have a long, static table of exoplanets.

Documentation:

- Jinja2 for loops

Step 4: Make the Table Interactive

Action:

A static table isn't very useful for analysis. We'll use a simple yet powerful JavaScript library to add client-side sorting and filtering.

1. **Choose a library.** A fantastic, lightweight choice is Simple-DataTables. It's easy to set up and has no dependencies like jQuery.
2. **Add the library to base.html**. In your base template, include the CSS in the <head> and the JavaScript file just before the closing </body> tag by linking to their CDN versions.
3. **Initialize the library in dashboard.html**. Add a <script> tag at the bottom of your dashboard template. Inside, write the few lines of JavaScript needed to activate Simple-DataTables on your HTML table. The library's documentation will show you exactly how to do this (it's usually just one line of code).

Why: Performing sorting and filtering on the **client-side** (in the user's browser) is vastly more efficient for a dataset of this size than sending a request back to your Flask server every time a user clicks a column header. The user gets an instant, snappy experience.

Step 5: Update Site Navigation

File to Modify: app/templates/base.html
Action:
Now that you have a dashboard, login, and logout, update your main navigation bar.

1. Use if current_user.is_authenticated blocks in the template.
2. If the user is logged in, show links to the "Dashboard" and "Logout".
3. If they are not logged in, show links to "Login" and "Register".
4. If current_user.role == 'admin', you can also show a link to the "Admin Panel".

Why: This creates a clean and logical user experience, showing users only the options that are relevant to them based on their authentication status and role.
After completing this task, you will have the centerpiece of your application: a functional, interactive data dashboard. Good luck!