Here's how you can structure the start of your project:

**1. Clarify the Scope**

* **Review Your Objectives**: Revisit the aims outlined in your progress report to ensure you understand the goals. For instance:
  + Consolidate SCOP, CATH, and DynDom data into an SQL database.
  + Compare structural and dynamic domains to understand their functional roles in proteins.
* **Set Priorities**: Decide if your initial focus will be on database integration, web server design, or domain comparison.

**2. Start with Database Integration**

* **Design Your Database**:
  + Create a schema to integrate SCOP, CATH, and DynDom data.
  + Define relationships (e.g., protein ID, domain boundaries, structural annotations).
* **Import Data**:
  + Begin by loading data from each database into your SQL tables.
  + Validate imported data to ensure it aligns with the schema.
* **Write Basic Queries**:
  + Test SQL queries to retrieve information (e.g., "List all domains for protein X").

**3. Conduct Initial Data Analysis**

* **Define Questions**:
  + What metrics will you compare (e.g., size, boundaries, overlaps)?
  + Are you looking for correlations between structural and dynamic definitions?
* **Run Comparative Queries**:
  + Write SQL queries to identify overlaps or mismatches between domains from SCOP, CATH, and DynDom.
* **Generate Basic Statistics**:
  + Use tools like Python or R to calculate and visualize metrics (e.g., percentage overlap between domain types).

**4. Prototype the Web Server**

* **Set Up Basic Infrastructure**:
  + Start with a simple Flask or Django app (Python frameworks).
  + Include a basic interface for querying your SQL database.
* **Add Visualization**:
  + Incorporate simple data visualization using libraries like Matplotlib, Plotly, or D3.js.

**5. Develop a Work Plan**

* **Short-Term Goals**:
  + Complete database integration.
  + Perform initial comparative analysis of a small dataset.
* **Medium-Term Goals**:
  + Expand analysis to larger datasets.
  + Develop a functional prototype of the web server.
* **Long-Term Goals**:
  + Optimize server performance.
  + Add advanced features like dynamic predictions using DynDom algorithms.

**6. Use the Gantt Chart**

* Your report includes a Gantt chart. Use this as a guide for structuring your tasks week-by-week.

SQL

**Step 1: Database Schema Design**

Let's design a schema that accommodates data from SCOP, CATH, and DynDom. We’ll define tables for proteins, domains, and annotations (structural, functional, dynamic).

1. **Protein Table**: This table will store basic information about proteins.
2. **Domain Table**: This will hold the domain information, including boundary positions and type.
3. **Annotations Table**: This will store additional metadata about domains, including structural or dynamic properties.
4. **Protein-Domain Relationship**: A join table to represent the many-to-many relationship between proteins and domains.

**SQL Schema (PostgreSQL Example)**

-- Create Protein Table

CREATE TABLE proteins (

protein\_id SERIAL PRIMARY KEY,

protein\_name VARCHAR(255),

sequence TEXT

);

-- Create Domain Table

CREATE TABLE domains (

domain\_id SERIAL PRIMARY KEY,

domain\_name VARCHAR(255),

domain\_type VARCHAR(100), -- e.g., structural, dynamic

start\_position INT,

end\_position INT

);

-- Create Protein-Domain Relationship Table

CREATE TABLE protein\_domains (

protein\_id INT REFERENCES proteins(protein\_id),

domain\_id INT REFERENCES domains(domain\_id),

PRIMARY KEY (protein\_id, domain\_id)

);

-- Create Annotations Table

CREATE TABLE annotations (

annotation\_id SERIAL PRIMARY KEY,

domain\_id INT REFERENCES domains(domain\_id),

annotation\_type VARCHAR(100), -- e.g., "structural", "dynamic"

annotation\_value TEXT

);

**Step 2: Import Data**

After creating the schema, you'll need to import the data from SCOP, CATH, and DynDom. For this example, I'll show how you might load data using SQL COPY statements (assuming your data is in CSV format).

Here is an example of how to load SCOP data into the proteins and domains tables:

sql

CopyEdit

-- Load SCOP data into proteins table

COPY proteins(protein\_name, sequence)

FROM '/path/to/scop\_proteins.csv' DELIMITER ',' CSV HEADER;

-- Load SCOP data into domains table

COPY domains(domain\_name, domain\_type, start\_position, end\_position)

FROM '/path/to/scop\_domains.csv' DELIMITER ',' CSV HEADER;

-- Load Protein-Domain relationships

COPY protein\_domains(protein\_id, domain\_id)

FROM '/path/to/scop\_protein\_domains.csv' DELIMITER ',' CSV HEADER;

Repeat a similar process for the CATH and DynDom datasets, adjusting the columns to match your data.

**Step 3: Query to Test Integration**

Once your data is imported, you can write queries to validate the data:

sql

CopyEdit

-- Query to list all domains for a given protein (protein\_name='ProteinA')

SELECT p.protein\_name, d.domain\_name

FROM proteins p

JOIN protein\_domains pd ON p.protein\_id = pd.protein\_id

JOIN domains d ON d.domain\_id = pd.domain\_id

WHERE p.protein\_name = 'ProteinA';

**Step 4: Validation**

Validate that the imported data is correct by running some basic queries, such as:

1. List all domains in a given protein.
2. Identify overlapping domains (e.g., same start-end positions).
3. Count the number of domains per protein.

BACKEND SERVER

Your current HTML structure looks great, and I see you're focusing on two main features: searching for protein domains and comparing structural and dynamic domains. To integrate this with your database and make it more interactive, you’ll need to implement the following:

1. **Backend Server (Python + Flask)**: You will need a Flask server to interact with your SQL database, handle user searches, and return results.
2. **JavaScript to Fetch Data**: On the frontend, you can use JavaScript to make API calls to the Flask server and display the data (e.g., domain comparisons, protein details).

Let’s integrate the front-end HTML you provided with Flask (as a backend server) and implement some of the interactive functionality.

**Step 1: Flask Backend Setup**

You'll need to install Flask first (if you haven't already):

bash

CopyEdit

pip install flask

Create a app.py file to handle requests from the frontend.

python

CopyEdit

from flask import Flask, render\_template, request, jsonify

import psycopg2

app = Flask(\_\_name\_\_)

# Database connection (use your own credentials)

def get\_db\_connection():

conn = psycopg2.connect(

host="localhost",

database="protein\_db", # Replace with your database name

user="your\_username", # Replace with your username

password="your\_password" # Replace with your password

)

return conn

@app.route('/')

def index():

return render\_template('index.html') # your HTML file

@app.route('/search', methods=['GET'])

def search():

protein\_name = request.args.get('protein-name')

if not protein\_name:

return jsonify({'error': 'No protein name provided'}), 400

conn = get\_db\_connection()

cur = conn.cursor()

# Query to fetch protein and domain information based on the protein name

cur.execute("""

SELECT p.protein\_name, d.domain\_name, d.start\_position, d.end\_position

FROM proteins p

JOIN protein\_domains pd ON p.protein\_id = pd.protein\_id

JOIN domains d ON pd.domain\_id = d.domain\_id

WHERE p.protein\_name = %s

""", (protein\_name,))

result = cur.fetchall()

cur.close()

conn.close()

if result:

return jsonify({'protein': protein\_name, 'domains': result})

else:

return jsonify({'error': 'Protein not found'}), 404

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

This server does the following:

* When you access the root URL (/), it renders the index.html file (your existing webpage).
* When a user submits the search form, it makes a GET request to the /search endpoint with the protein name or ID, and the server responds with the relevant protein domain data.

**Step 2: Frontend JavaScript Update**

You need JavaScript code to send a request to the backend when the user submits the search form. Update your script.js file to send AJAX requests to the Flask server.

javascript

CopyEdit

document.getElementById("search-form").addEventListener("submit", function (event) {

event.preventDefault(); // Prevent form from refreshing the page

const proteinName = document.getElementById("protein-name").value;

// Send AJAX request to Flask backend

fetch(`/search?protein-name=${proteinName}`)

.then(response => response.json())

.then(data => {

if (data.error) {

alert(data.error); // Show error message if protein not found

} else {

displayProteinData(data); // Call function to display results

}

})

.catch(error => {

console.error('Error fetching data:', error);

});

});

function displayProteinData(data) {

const proteinName = data.protein;

const domains = data.domains;

const structuralDomainPanel = document.getElementById("structural-domain");

const dynamicDomainPanel = document.getElementById("dynamic-domain");

// Clear existing content

structuralDomainPanel.innerHTML = "<h3>Structural Domain</h3>";

dynamicDomainPanel.innerHTML = "<h3>Dynamic Domain</h3>";

// Display domains (for now, just list them)

domains.forEach(domain => {

const domainElement = document.createElement("p");

domainElement.innerHTML = `${domain[1]} (Position: ${domain[2]} - ${domain[3]})`;

if (domain[1].includes("Structural")) {

structuralDomainPanel.appendChild(domainElement);

} else {

dynamicDomainPanel.appendChild(domainElement);

}

});

}

**Step 3: Update HTML for Dynamic Content**

Now, update the HTML slightly so the JavaScript can dynamically display the domain results:

html

CopyEdit

<!-- Structural Domain Visualization -->

<div id="structural-domain" class="domain-panel">

<h3>Structural Domain</h3>

<p>Visualization of the structural domain for the selected protein will appear here.</p>

<div class="viewer-placeholder">3D Viewer Placeholder</div>

</div>

<!-- Dynamic Domain Visualization -->

<div id="dynamic-domain" class="domain-panel">

<h3>Dynamic Domain</h3>

<p>Visualization of the dynamic domain for the selected protein will appear here.</p>

<div class="viewer-placeholder">Dynamic Viewer Placeholder</div>

</div>

This will let you dynamically populate the content of the structural and dynamic domain sections based on the search results.

**Step 4: Run the Flask App**

Make sure your Flask app is running:

bash

CopyEdit

python app.py

Now, when you visit http://localhost:5000 in your browser, you can search for proteins, and the corresponding domain data will be fetched from your database and displayed on the page.

**Next Steps:**

* **3D Visualization**: Implement 3D protein structure visualization using tools like **PyMOL**, **JSmol**, or **3Dmol.js**. This can be integrated into the placeholders.
* **Styling**: Use CSS for a polished look and responsive layout. You may already have a style.css for this.
* **Error Handling**: Implement more detailed error handling on both frontend and backend to improve user experience.

Let me know if you need more specific features or adjustments!

**PG ADMIN**

Great! Since you're using PGAdmin, the setup for your PostgreSQL database will be similar, but with PGAdmin's GUI interface for managing the database. Below are the steps to connect and manage your PostgreSQL database using PGAdmin.

**Steps to Set Up the Database in PGAdmin**

1. **Open PGAdmin**:
   * Open PGAdmin on your computer and connect to your PostgreSQL server.
2. **Create a New Database**:
   * In PGAdmin, navigate to the "Servers" section, and right-click on your server.
   * Click on **Create** > **Database**.
   * Name the database (e.g., protein\_db) and click **Save**.
3. **Create Tables for Proteins and Domains**:
   * After creating the database, right-click on the database name and select **Query Tool**.
   * Run the following SQL commands to create the necessary tables:

sql

CopyEdit

-- Create table for proteins

CREATE TABLE proteins (

protein\_id SERIAL PRIMARY KEY,

protein\_name VARCHAR(255) UNIQUE NOT NULL

);

-- Create table for domains

CREATE TABLE domains (

domain\_id SERIAL PRIMARY KEY,

domain\_name VARCHAR(255) NOT NULL,

start\_position INT NOT NULL,

end\_position INT NOT NULL

);

-- Create a junction table to link proteins and domains

CREATE TABLE protein\_domains (

protein\_id INT REFERENCES proteins(protein\_id) ON DELETE CASCADE,

domain\_id INT REFERENCES domains(domain\_id) ON DELETE CASCADE,

PRIMARY KEY (protein\_id, domain\_id)

);

1. **Insert Data**:
   * You can insert data manually through PGAdmin's **Query Tool**, or you can use CSV import if you have large datasets.
   * Example SQL for inserting data into the tables:

sql

CopyEdit

-- Insert sample protein

INSERT INTO proteins (protein\_name) VALUES ('1ubq');

-- Insert sample domains

INSERT INTO domains (domain\_name, start\_position, end\_position)

VALUES ('Structural Domain', 1, 50),

('Dynamic Domain', 51, 100);

-- Link protein to domains

INSERT INTO protein\_domains (protein\_id, domain\_id)

VALUES ((SELECT protein\_id FROM proteins WHERE protein\_name = '1ubq'),

(SELECT domain\_id FROM domains WHERE domain\_name = 'Structural Domain'));

1. **Test the Database Connection in Flask**:
   * Ensure that your Flask app can connect to the PostgreSQL database.
   * Use the correct credentials for the connection, like this:

python

CopyEdit

conn = psycopg2.connect(

host="localhost",

database="protein\_db", # The name of your database

user="your\_username", # Your PGAdmin username

password="your\_password" # Your PGAdmin password

)

1. **Running Your Flask App**:
   * After setting up your database and ensuring your Flask app is connected to PGAdmin, run the Flask app using:

bash

CopyEdit

python app.py

Now, your Flask app will be connected to your PostgreSQL database through PGAdmin, and it will serve the data to your website based on the protein and domain search functionality.

Let me know if you need help with any of the steps!

CONNECTING DATABASE DB.JS

const { Pool } = require("pg");

const pool = new Pool({

user: "your\_username",

host: "localhost",

database: "your\_database",

password: "your\_password",

port: 5432,

});

module.exports = pool;

CATH DATABASE

CREATE TABLE cath\_domains (

cath\_id TEXT PRIMARY KEY,

class INT NOT NULL,

architecture INT NOT NULL,

topology INT NOT NULL,

homology INT NOT NULL,

description TEXT

);

DYN DOM DATABASE

CREATE TABLE dyndom\_domains (

pdb\_code TEXT NOT NULL,

chain\_id TEXT NOT NULL,

domain\_id TEXT NOT NULL,

start\_residue INT NOT NULL,

end\_residue INT NOT NULL,

PRIMARY KEY (pdb\_code, chain\_id, domain\_id)

);

Residue mapping database

CREATE TABLE residue\_mapping (

id SERIAL PRIMARY KEY,

pdb\_code TEXT NOT NULL,

chain\_id TEXT NOT NULL,

residue\_number INT NOT NULL,

cath\_id TEXT NOT NULL,

domain\_source TEXT CHECK (domain\_source IN ('CATH', 'DynDom'))

);

SERVER.JS

const express = require("express");

const cors = require("cors");

const { Pool } = require("pg");

const app = express();

const PORT = 3000;

const pool = new Pool({

user: "your\_username",

host: "localhost",

database: "your\_database",

password: "your\_password",

port: 5432,

});

app.use(cors());

app.use(express.json());

app.use(express.static("public"));

// Get color-coded sequence based on PDB & Chain ID

app.get("/api/domains/:pdbCode/:chainId", async (req, res) => {

try {

const { pdbCode, chainId } = req.params;

const query = `

SELECT residue\_number, cath\_id, domain\_source

FROM residue\_mapping

WHERE pdb\_code = $1 AND chain\_id = $2

ORDER BY residue\_number;

`;

const result = await pool.query(query, [pdbCode, chainId]);

if (result.rows.length === 0) {

return res.status(404).json({ error: "No data found for this PDB Code and Chain ID." });

}

res.json(result.rows);

} catch (err) {

console.error(err);

res.status(500).json({ error: "Database error" });

}

});

app.listen(PORT, () => console.log(`Server running on [http://localhost:${PORT}`)](http://localhost:$%7bPORT%7d%60)));