Student Sport Portfolio - Working Documentation

This document provides a comprehensive guide to setting up, running, and understanding the Student Sport Portfolio application.

Table of Contents

1. Introduction

The Student Sport Portfolio is a web application designed to help students track and manage their sports achievements and participation. It provides a personalized dashboard for students to record their tournament entries, view their statistics (wins, total tournaments, win rate, sports played), and update their profile. An administrative panel is also included for overall monitoring of student activities and sports statistics.

The application is built using Flask, a Python web framework, and SQLite3 as its database management system for simplicity and ease of setup. Frontend is developed using HTML, CSS (with a custom DuckDuckGo-inspired design system), and Bootstrap for responsive UI.

2. Features

- **Student Authentication:** Secure login and signup for students using USN and Date of Birth.
- **Admin Authentication:** Separate login for administrators to manage the system.
- **Student Dashboard:**
  - Overview of personal sports statistics (wins, total tournaments, win rate, sports played).
  - Display of student profile information.
  - List of all recorded tournament entries.
  - Ability to add new tournament entries.
  - Ability to edit and delete existing tournament entries.
- **Tournament Management:** Comprehensive forms for recording tournament details including sport, date, level, event type, opponents, scores, outcome, and awards.

- **Dynamic UI:** Responsive design using Bootstrap and custom CSS, with a theme switcher for light/dark mode.
- **Admin Dashboard:**
  - View total registered students.
  - Pie chart visualizing student distribution across different sports.
  - List of tournaments count per sport.
  - Detailed list of all student profiles with search and sort functionalities.
- **SQLite3 Database:** Lightweight and file-based database for easy setup.

3. Prerequisites

Before running the application, ensure you have the following installed:

- **Python 3.x**
- **pip** (Python package installer)

4. Installation and Setup

Follow these steps to get the application up and running on your local machine.

4.1. Clone the Repository

First, clone the project repository to your local machine. (Assuming the user has the repository).

```
# If you have a Git repository, you would clone it here.
# Example: git clone <repository_url>
# cd student-sport-portfolio
```

4.2. Install Dependencies

Navigate to the project directory (where app.py is located) and install the required Python packages using pip:

```
pip install Flask
```

4.3. Database Initialization

The application uses SQLite3, which is a file-based database. The student_sport_portfolio.db file will be created automatically when the application runs for the first time. The init_db() function in app.py handles the creation of tables and insertion of default sports.

4.4. Running the Application

To start the Flask development server, execute the app.py file:

```
python app.py
```

You should see output similar to this:

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: XXX-XXX-XXX
```

Open your web browser and navigate to http://127.0.0.1:5000 (or the address shown in your console).

5. Application Structure

The project follows a standard Flask application structure:

```
student-sport-portfolio/
 ├── app.py              # Main Flask application logic
 ├── student_sport_portfolio.db # SQLite database file (created on first run)
 ├── static/             # Static files (CSS, JS, images)
 │   └── style.css        # Custom CSS for styling
 │   └── logo.svg         # Application logo
 ├── templates/           # HTML templates
 │   ├── login.html        # Student login page
 │   ├── signup.html       # Student registration page
 │   ├── dashboard.html     # Student dashboard
 │   ├── add_edit_tournament.html # Form for adding/editing tournaments
 │   ├── admin_login.html   # Admin login page
 │   ├── set_admin_credentials.html # Admin setup page
 │   └── admin_dashboard.html # Admin dashboard
 └── README.md            # This documentation file
```

6. File Explanations

6.1. app.py

This is the core Python backend file for the Flask application. It handles routing, database interactions, session management, and rendering HTML templates.

Key Components:

- **Flask Setup:**
```
from flask import Flask, render_template, request, redirect, url_for, flash, session, jsonify
import sqlite3
from datetime import datetime, date
import hashlib
import os

app = Flask(__name__)
app.secret_key = 'your-secret-key-here' # IMPORTANT: Change this!
```

Initializes the Flask app and sets a secret key for secure session management (essential for flash messages and session data).

- init_db() function:
  This function is called when the application starts (if __name__ == '__main__':). It connects to student_sport_portfolio.db (creating it if it doesn't exist) and executes SQL commands to create four tables:
  - Students: Stores student personal details.
  - Sports: Stores a predefined list of sports.
  - Tournaments: Stores details of each tournament entry, with foreign keys linking to Students and Sports.
  - Admin: Stores admin login credentials.
    It also populates the Sports table with a default list of sports using INSERT OR IGNORE to prevent duplicates on subsequent runs.

```
def init_db():
  conn = sqlite3.connect('student_sport_portfolio.db')
  cursor = conn.cursor()
  # Create Students table
  cursor.execute('''
    CREATE TABLE IF NOT EXISTS Students (
      StudentID INTEGER PRIMARY KEY AUTOINCREMENT,
      USN TEXT UNIQUE NOT NULL,
      FirstName TEXT NOT NULL,
      LastName TEXT NOT NULL,
      DateOfBirth DATE NOT NULL,
      Email TEXT NOT NULL,
      CollegeJoiningYear INTEGER,
      CollegeEndingYear INTEGER,
      Branch TEXT,
      Section TEXT,
      PhoneNumber TEXT,
      AgentName TEXT,
      AgentPhoneNumber TEXT,
      CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
      UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')
# Create Sports table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Sports (
        SportID INTEGER PRIMARY KEY AUTOINCREMENT,
        SportName TEXT UNIQUE NOT NULL
    )
''')
# Create Tournaments table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Tournaments (
        TournamentID INTEGER PRIMARY KEY AUTOINCREMENT,
        StudentID INTEGER,
        SportID INTEGER,
        TournamentName TEXT NOT NULL,
        TournamentDate DATE NOT NULL,
        LeagueLevel TEXT NOT NULL,
        EventType TEXT NOT NULL,
        TeamName TEXT,
        TeamRole TEXT,
        Opponents TEXT NOT NULL,
        CoachName TEXT,
        StudentScore TEXT,
        OpponentScore TEXT,
        Outcome TEXT NOT NULL,
        AchievementsAwards TEXT,
        Notes TEXT,
        CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (StudentID) REFERENCES Students (StudentID),
        FOREIGN KEY (SportID) REFERENCES Sports (SportID)
    )
''')
# Create Admin table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS Admin (
        AdminID INTEGER PRIMARY KEY AUTOINCREMENT,
        Username TEXT UNIQUE NOT NULL,
        PasswordHash TEXT NOT NULL,
        CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')
# Insert default sports
sports = [
    'Cricket', 'Football', 'Weightlifting', 'Powerlifting', 'Tennis',
    'Table Tennis', 'Basketball', 'Kho Kho', 'Kabaddi', 'Volleyball',
    'Athletics', 'Badminton', 'Chess'
]
for sport in sports:
    cursor.execute('INSERT OR IGNORE INTO Sports (SportName) VALUES (?)', (sport,))
conn.commit()
conn.close()
```

- get_db_connection() function:
  A helper function to establish a new database connection. It sets conn.row_factory = sqlite3.Row, which allows accessing query results like dictionaries (e.g., row['ColumnName']) instead of just by index.
  ```
  def get_db_connection():
      conn = sqlite3.connect('student_sport_portfolio.db')
      conn.row_factory = sqlite3.Row
      return conn
  ```

- hash_password(password) function:
  Uses hashlib.sha256 to hash passwords. This is a basic hashing mechanism. For production environments, stronger, adaptive hashing algorithms like bcrypt are recommended.
  ```
  def hash_password(password):
      return hashlib.sha256(password.encode()).hexdigest()
  ```

- row_to_dict(row) and rows_to_dict_list(rows) functions:
  Utility functions to convert SQLite Row objects (which behave like tuples) into dictionaries, making data easier to work with in Python.

```
def row_to_dict(row):
    """Convert SQLite Row object to dictionary"""
    if row is None:
        return None
    return dict(row)

def rows_to_dict_list(rows):
    """Convert list of SQLite Row objects to list of dictionaries"""
    return [dict(row) for row in rows]
```

- calculate_student_stats(student_id) function:
  Queries the database to compute statistics for a specific student, including total tournaments, wins, unique sports played, and win percentage.

```
def calculate_student_stats(student_id):
    """Calculate statistics for a specific student"""
    conn = get_db_connection()

    # Get total tournaments
    total_tournaments = conn.execute(
        'SELECT COUNT(*) as count FROM Tournaments WHERE StudentID = ?',
        (student_id,)
    ).fetchone()['count']

    # Get wins
    total_wins = conn.execute(
        'SELECT COUNT(*) as count FROM Tournaments WHERE StudentID = ? AND
Outcome = "Win"',
        (student_id,)
    ).fetchone()['count']
    # Get unique sports played
    sports_played = conn.execute(
        'SELECT COUNT(DISTINCT SportID) as count FROM Tournaments WHERE
StudentID = ?',
        (student_id,)
    ).fetchone()['count']

    # Calculate win percentage
    win_percentage = round((total_wins / total_tournaments * 100) if
total_tournaments > 0 else 0, 1)

    conn.close()

    return {
        'total_tournaments': total_tournaments,
        'total_wins': total_wins,
        'sports_played': sports_played,
        'win_percentage': win_percentage
    }
```

- check_admin_exists() function:
  Determines if an admin account has already been created in the Admin table. This is crucial for controlling access to the set_admin_credentials route.

```
def check_admin_exists():
    """Check if admin credentials exist"""
    conn = get_db_connection()
    admin_count = conn.execute('SELECT COUNT(*) as count FROM
Admin').fetchone()['count']
    conn.close()
    return admin_count > 0
```

- Routes (@app.route(...)):
  Each @app.route decorator defines a URL endpoint and the Python function that handles requests to that URL.
  - /: Redirects to /login.
  - /login: Handles student login (GET for form, POST for authentication).

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        usn = request.form['usn'].upper()
        dob = request.form['dob']

        conn = get_db_connection()
        student = conn.execute(
            'SELECT * FROM Students WHERE USN = ? AND DateOfBirth = ?',
            (usn, dob)
        ).fetchone()
        conn.close()

        if student:
            session['student_id'] = student['StudentID']
            session['student_name'] = f"{student['FirstName']} {student['LastName']}"
            flash('Login successful!', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Invalid USN or Date of Birth. Please try again.', 'danger')

    return render_template('login.html')
```

  - /signup: Handles student registration (GET for form, POST for creating account).

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        try:
            usn = request.form['usn'].upper()
```

```
first_name = request.form['first_name']
last_name = request.form['last_name']
dob = request.form['dob']
email = request.form['email']
college_joining_year = int(request.form['college_joining_year'])
college_ending_year = int(request.form['college_ending_year'])
branch = request.form['branch']
section = request.form['section']
phone_number = request.form['phone_number']
agent_name = request.form.get('agent_name', '')
agent_phone_number = request.form.get('agent_phone_number', '')

conn = get_db_connection()

existing_student = conn.execute(
  'SELECT USN FROM Students WHERE USN = ?', (usn,)
).fetchone()

if existing_student:
  flash('A student with this USN already exists.', 'danger')
  conn.close()
  return render_template('signup.html',
             form_data=request.form,
             current_year=datetime.now().year)

conn.execute('''
  INSERT INTO Students (
    USN, FirstName, LastName, DateOfBirth, Email,
    CollegeJoiningYear, CollegeEndingYear, Branch, Section,
    PhoneNumber, AgentName, AgentPhoneNumber
  ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
''', (
  usn, first_name, last_name, dob, email,
  college_joining_year, college_ending_year, branch, section,
  phone_number, agent_name, agent_phone_number
))

conn.commit()
```

```
            conn.close()

            flash('Account created successfully! You can now log in.', 'success')
            return redirect(url_for('login'))

        except Exception as e:
            flash(f'Error creating account: {str(e)}', 'danger')
            return render_template('signup.html',
                                   form_data=request.form,
                                   current_year=datetime.now().year)

    return render_template('signup.html', current_year=datetime.now().year)
```

- /dashboard: Displays the student's personal dashboard, fetching their profile and tournament data. Requires login.

```
@app.route('/dashboard')
def dashboard():
    if 'student_id' not in session:
        flash('Please log in to access the dashboard.', 'warning')
        return redirect(url_for('login'))

    student_id = session['student_id']
    conn = get_db_connection()

    student = conn.execute(
        'SELECT * FROM Students WHERE StudentID = ?',
        (student_id,)
    ).fetchone()

    tournaments = conn.execute('''
        SELECT t.*, s.SportName
        FROM Tournaments t
        JOIN Sports s ON t.SportID = s.SportID
        WHERE t.StudentID = ?
        ORDER BY t.TournamentDate DESC
    ''', (student_id,)).fetchall()

    conn.close()

    stats = calculate_student_stats(student_id)

    formatted_tournaments = []
    for tournament in tournaments:
        tournament_dict = dict(tournament)
        if tournament_dict['TournamentDate']:
            date_obj = datetime.strptime(tournament_dict['TournamentDate'], '%Y-%m-
%d')
            tournament_dict['TournamentDate'] = date_obj.strftime('%d/%m/%Y')
        formatted_tournaments.append(tournament_dict)

    return render_template('dashboard.html',
                student=student,
                tournaments=formatted_tournaments,
                stats=stats)
```

- /add_tournament: Allows students to add new tournament entries.

```
@app.route('/add_tournament', methods=['GET', 'POST'])
def add_tournament():
    if 'student_id' not in session:
        flash('Please log in to access this page.', 'warning')
        return redirect(url_for('login'))

    if request.method == 'POST':
        try:
            student_id = session['student_id']
            sport_id = request.form['sport_id']
            tournament_name = request.form['tournament_name']
            tournament_date = request.form['tournament_date']
            league_level = request.form['league_level']
            event_type = request.form['event_type']
            team_name = request.form.get('team_name', '')
            team_role = request.form.get('team_role', '')
            opponents = request.form['opponents']
            coach_name = request.form.get('coach_name', '')
            student_score = request.form.get('student_score', '')
            opponent_score = request.form.get('opponent_score', '')
            outcome = request.form['outcome']
            achievements_awards = request.form.get('achievements_awards', '')
            notes = request.form.get('notes', '')

            conn = get_db_connection()
            conn.execute('''
                INSERT INTO Tournaments (
                    StudentID, SportID, TournamentName, TournamentDate, LeagueLevel,
                    EventType, TeamName, TeamRole, Opponents, CoachName,
                    StudentScore, OpponentScore, Outcome, AchievementsAwards, Notes
                ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
            ''', (
                student_id, sport_id, tournament_name, tournament_date, league_level,
                event_type, team_name, team_role, opponents, coach_name,
                student_score, opponent_score, outcome, achievements_awards, notes
            ))
            conn.commit()
            conn.close()

            flash('Tournament entry added successfully!', 'success')
            return redirect(url_for('dashboard'))

        except Exception as e:
            flash(f'Error adding tournament: {str(e)}', 'danger')

    conn = get_db_connection()
    sports = conn.execute('SELECT * FROM Sports ORDER BY SportName').fetchall()
    conn.close()

    return render_template('add_edit_tournament.html',
                sports=sports,
                is_edit=False)
```

- /edit_tournament/<int:tournament_id>: Allows students to edit existing tournament entries.

```
@app.route('/edit_tournament/<int:tournament_id>', methods=['GET', 'POST'])
def edit_tournament(tournament_id):
  if 'student_id' not in session:
    flash('Please log in to access this page.', 'warning')
    return redirect(url_for('login'))

  student_id = session['student_id']
  conn = get_db_connection()

  tournament = conn.execute(
    'SELECT * FROM Tournaments WHERE TournamentID = ? AND StudentID = ?',
    (tournament_id, student_id)
  ).fetchone()

  if not tournament:
    flash('Tournament not found or access denied.', 'danger')
    conn.close()
    return redirect(url_for('dashboard'))

  if request.method == 'POST':
    try:
      sport_id = request.form['sport_id']
      tournament_name = request.form['tournament_name']
      tournament_date = request.form['tournament_date']
      league_level = request.form['league_level']
      event_type = request.form['event_type']
      team_name = request.form.get('team_name', '')
      team_role = request.form.get('team_role', '')
      opponents = request.form['opponents']
      coach_name = request.form.get('coach_name', '')
      student_score = request.form.get('student_score', '')
      opponent_score = request.form.get('opponent_score', '')
```

```
        outcome = request.form['outcome']
        achievements_awards = request.form.get('achievements_awards', '')
        notes = request.form.get('notes', '')

        conn.execute('''
          UPDATE Tournaments SET
            SportID = ?, TournamentName = ?, TournamentDate = ?, LeagueLevel = ?,
            EventType = ?, TeamName = ?, TeamRole = ?, Opponents = ?, CoachName
= ?,
            StudentScore = ?, OpponentScore = ?, Outcome = ?,
AchievementsAwards = ?,
            Notes = ?, UpdatedAt = CURRENT_TIMESTAMP
          WHERE TournamentID = ? AND StudentID = ?
        ''', (
          sport_id, tournament_name, tournament_date, league_level,
          event_type, team_name, team_role, opponents, coach_name,
          student_score, opponent_score, outcome, achievements_awards, notes,
          tournament_id, student_id
        ))
        conn.commit()
        conn.close()

        flash('Tournament entry updated successfully!', 'success')
        return redirect(url_for('dashboard'))

      except Exception as e:
        flash(f'Error updating tournament: {str(e)}', 'danger')

    sports = conn.execute('SELECT * FROM Sports ORDER BY SportName').fetchall()

    display_tournament_date = tournament['TournamentDate']

    conn.close()

    return render_template('add_edit_tournament.html',
              sports=sports,
              tournament=tournament,
              display_tournament_date=display_tournament_date,
            is_edit=True)
```

- /delete_tournament/<int:tournament_id>: Handles AJAX requests to delete a tournament entry.

```
@app.route('/delete_tournament/<int:tournament_id>', methods=['POST'])
def delete_tournament(tournament_id):
  if 'student_id' not in session:
    return jsonify({'success': False, 'message': 'Not logged in'}), 401

  student_id = session['student_id']

  try:
    conn = get_db_connection()

    tournament = conn.execute(
      'SELECT TournamentID FROM Tournaments WHERE TournamentID = ? AND
StudentID = ?',
      (tournament_id, student_id)
    ).fetchone()

    if not tournament:
      conn.close()
      return jsonify({'success': False, 'message': 'Tournament not found or access
denied'}), 404

    conn.execute(
      'DELETE FROM Tournaments WHERE TournamentID = ? AND StudentID = ?',
      (tournament_id, student_id)
    )
    conn.commit()
    conn.close()

    return jsonify({'success': True, 'message': 'Tournament deleted successfully'})

  except Exception as e:
        return jsonify({'success': False, 'message': str(e)}), 500
```

- /logout: Clears student session and redirects to login.

```
@app.route('/logout')
def logout():
    session.clear()
    flash('You have been logged out successfully.', 'info')
    return redirect(url_for('login'))
```

- /admin: Redirects to /admin_login or /set_admin_credentials based on admin existence.

```
@app.route('/admin')
def admin():
    """Redirect to admin login or setup based on whether admin exists"""
    if check_admin_exists():
        return redirect(url_for('admin_login'))
    else:
        return redirect(url_for('set_admin_credentials'))
```

- /admin_login: Handles admin login.

```
@app.route('/admin_login', methods=['GET', 'POST'])
def admin_login():
    if not check_admin_exists():
        flash('No admin credentials found. Please set up admin credentials first.',
'warning')
        return redirect(url_for('set_admin_credentials'))

    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        password_hash = hash_password(password)

        conn = get_db_connection()
        admin = conn.execute(
            'SELECT * FROM Admin WHERE Username = ? AND PasswordHash = ?',
            (username, password_hash)
        ).fetchone()
        conn.close()

        if admin:
            session['admin_id'] = admin['AdminID']
            session['admin_username'] = admin['Username']
            flash('Admin login successful!', 'success')
            return redirect(url_for('admin_dashboard'))
        else:
            flash('Invalid username or password.', 'danger')

    return render_template('admin_login.html')
```

- /admin_dashboard: Displays the admin dashboard with overall statistics and student list. Requires admin login.

```
@app.route('/admin_dashboard')
def admin_dashboard():
    if 'admin_id' not in session:
        flash('Please log in as admin to access this page.', 'warning')
        return redirect(url_for('admin_login'))

    conn = get_db_connection()

    total_students = conn.execute('SELECT COUNT(*) as count FROM
Students').fetchone()['count']

    students_per_sport_rows = conn.execute('''
        SELECT s.SportName, COUNT(DISTINCT t.StudentID) as student_count
        FROM Sports s
        LEFT JOIN Tournaments t ON s.SportID = t.SportID
        GROUP BY s.SportID, s.SportName
        ORDER BY student_count DESC
    ''').fetchall()
    students_per_sport = rows_to_dict_list(students_per_sport_rows)

    tournaments_per_sport_rows = conn.execute('''
        SELECT s.SportName, COUNT(t.TournamentID) as tournament_count
        FROM Sports s
        LEFT JOIN Tournaments t ON s.SportID = t.SportID
        GROUP BY s.SportID, s.SportName
        ORDER BY tournament_count DESC
    ''').fetchall()
    tournaments_per_sport = rows_to_dict_list(tournaments_per_sport_rows)

    students_rows = conn.execute('''
        SELECT
            s.*,
            COUNT(t.TournamentID) as total_matches_played,
            SUM(CASE WHEN t.Outcome = 'Win' THEN 1 ELSE 0 END) as total_wins,
            SUM(CASE WHEN t.Outcome = 'Loss' THEN 1 ELSE 0 END) as total_losses,
            GROUP_CONCAT(DISTINCT sp.SportName) as sports_played_list,
            MAX(t.UpdatedAt) as last_updated_at_raw
        FROM Students s
        LEFT JOIN Tournaments t ON s.StudentID = t.StudentID
        LEFT JOIN Sports sp ON t.SportID = sp.SportID
        GROUP BY s.StudentID
        ORDER BY s.USN
    ''').fetchall()

    conn.close()

    students_list = []
    for student_row in students_rows:
        student_dict = dict(student_row)

        if student_dict['last_updated_at_raw']:
            try:
            date_obj = datetime.strptime(student_dict['last_updated_at_raw'], '%Y-%m-
%d %H:%M:%S')
            student_dict['last_updated_at_display'] = date_obj.strftime('%d/%m/%Y')
            student_dict['last_updated_at_raw'] = date_obj.strftime('%d%m%Y')
          except:
            student_dict['last_updated_at_display'] = 'N/A'
            student_dict['last_updated_at_raw'] = '01011900'
        else:
          student_dict['last_updated_at_display'] = 'N/A'
          student_dict['last_updated_at_raw'] = '01011900'

        for key in ['total_matches_played', 'total_wins', 'total_losses']:
          if student_dict[key] is None:
            student_dict[key] = 0

        students_list.append(student_dict)

      return render_template('admin_dashboard.html',
                    total_students=total_students,
                    students_per_sport=students_per_sport,
                    tournaments_per_sport=tournaments_per_sport,
                    students=students_list)
```

- /set_admin_credentials: Allows initial setup of admin credentials. Only accessible if no admin exists.

```
@app.route('/set_admin_credentials', methods=['GET', 'POST'])
def set_admin_credentials():
  if check_admin_exists():
    flash('Admin credentials already set. Please use the login page.', 'info')
    return redirect(url_for('admin_login'))

  if request.method == 'POST':
    username = request.form['username']
    password = request.form['password']
    confirm_password = request.form['confirm_password']
```

```
        if password != confirm_password:
            flash('Passwords do not match.', 'danger')
            return render_template('set_admin_credentials.html')

        if len(password) < 6:
            flash('Password must be at least 6 characters long.', 'danger')
            return render_template('set_admin_credentials.html')

        password_hash = hash_password(password)

        try:
            conn = get_db_connection()
            conn.execute(
                'INSERT INTO Admin (Username, PasswordHash) VALUES (?, ?)',
                (username, password_hash)
            )
            conn.commit()
            conn.close()

            flash('Admin credentials set successfully! You can now log in.', 'success')
            return redirect(url_for('admin_login'))

        except Exception as e:
            flash(f'Error setting admin credentials: {str(e)}', 'danger')

    return render_template('set_admin_credentials.html')
```

- /admin_logout: Clears admin session and redirects to admin login.

```
@app.route('/admin_logout')
def admin_logout():
    session.pop('admin_id', None)
    session.pop('admin_username', None)
    flash('Admin logged out successfully.', 'info')
    return redirect(url_for('admin_login'))
```

6.2. style.css

This CSS file defines the visual design system for the entire application. It's built with a "DuckDuckGo-inspired" aesthetic, emphasizing clean lines, clear typography, and a balanced color palette.

Key Features:

- **CSS Variables (:root):** Defines a comprehensive set of custom properties for colors (primary, secondary, success, danger, neutral, background, text, border, input), shadows, border radii, spacing, typography, and transitions. This makes theme changes and consistent styling easy.
- **Dark Theme (body.dark-theme):** Overrides many of the light theme CSS variables when the dark-theme class is applied to the <body> element, enabling a seamless dark mode.
- **Base Styles:** Resets default browser styles and applies global font, background, and text colors.
- **Component Styling:** Provides specific styles for:
  - navbar: Top navigation bar.
  - main-content, content-wrapper, page-header: Layout and typography for main content areas.
  - alerts-container, alert-modern: Custom flash message styling with icons.
  - card-modern: General styling for cards used across dashboards and forms.
  - profile-section, profile-grid, agent-section: Layout and appearance of student profile details.
  - section-header: Styling for section titles and subtitles.
  - btn-primary-modern, btn-secondary-modern, btn-action-secondary, btn-action-danger: Custom button styles with hover effects.
  - tournaments-grid, tournament-card: Specific styles for displaying individual tournament entries.

  - empty-state: Styling for messages when no data is available.
  - form-control, form-select, form-label, form-text: Enhanced styling for form inputs and labels.
  - auth-container, auth-card, auth-header, auth-icon, auth-title, auth-subtitle, auth-form, form-group, form-control-modern, form-actions, auth-links, auth-link-small: Dedicated styles for authentication pages (login, signup, admin setup).
  - dashboard-widgets, student-grid, student-card, student-stats-widgets: Specific styles for the admin dashboard and student statistics.
  - input-group, sort-buttons, sport-selection-grid, sport-icon-button: Styles for search, filter, and sport selection components.
- **Animations:** Defines a fadeInUp keyframe animation for subtle entry effects.
- **Responsive Design (@media queries):** Adjusts layouts, font sizes, and component arrangements for optimal viewing on various screen sizes (desktops, tablets, mobiles).
- **Accessibility (:focus-visible):** Ensures clear visual focus indicators for keyboard navigation.
- **Print Styles (@media print):** Hides non-essential UI elements (like navigation and action buttons) when the page is printed, and ensures cards break cleanly.

6.3. login.html

This HTML file provides the user interface for the student login page.

Key Components:

- **HTML Structure:** Standard HTML5 boilerplate with meta tags for character set and viewport.
- **External Resources:** Links to Bootstrap CSS, custom style.css, and Google Material Symbols Outlined font.
- **Navigation Bar:** A simple navbar with the application brand and a theme switcher button.
- **Login Card (auth-card):** A centrally aligned card containing:
  - A header with a login icon, title ("Welcome Back"), and subtitle.
  - A section for displaying Flask flash messages (success, danger, warning, info).

  - A login form with fields for USN and Date of Birth (which serves as the password).
  - A "Sign In" button.
  - Links for "New Student? Create Account" (to signup.html) and "Admin Login" (to admin_login.html).
- **JavaScript:**
  - Includes Bootstrap's JavaScript bundle for UI functionality.
  - **Theme Switcher Logic:** Handles saving and applying the user's preferred theme (light/dark) to localStorage.
  - **Input Focus Transitions:** Adds and removes a focused class to parent elements of form inputs when they gain/lose focus, enabling smooth visual effects defined in style.css.
- **Inline Styles:** Contains some authentication-specific styles that are also present in style.css. These could be removed from here if style.css is always guaranteed to be loaded after.

6.4. signup.html

This HTML file provides the user interface for the student registration (signup) page.

Key Components:

- **HTML Structure:** Similar to login.html, with standard HTML5 boilerplate and external resource links.
- **Navigation Bar:** Consistent navbar with brand and theme switcher.
- **Page Header:** A header with a title ("Create Your Account") and subtitle.
- **Flash Messages:** Area for displaying Flask flash messages.
- **Signup Form (card-modern):** A card containing a detailed registration form for students:
  - Fields for USN, First Name, Last Name, Date of Birth (used as password), Email, College Joining Year, College Ending Year, Branch, Section, and Phone Number.
  - A checkbox "I have an agent or manager" that dynamically toggles the visibility of Agent Name and Agent Phone Number fields.
  - A "Create Account" button.

  - A "Back to Login" button.
- **JavaScript:**
  - Includes Bootstrap's JavaScript bundle.
  - **Theme Switcher Logic:** Same as login.html for light/dark mode.
  - **Agent Details Toggle:** Manages the maxHeight CSS property of the agent_details_section based on the has_agent_checkbox state, creating a smooth expand/collapse effect. It also clears agent input values when the checkbox is unchecked.

6.5. dashboard.html

This HTML file represents the main student dashboard, displaying their profile, statistics, and tournament entries.

Key Components:

- **HTML Structure:** Standard HTML5 boilerplate with links to Bootstrap CSS, style.css, Material Symbols, and Font Awesome.
- **Navigation Bar:** Includes the application brand, a welcome message for the logged-in student, a theme switcher, and a logout button.
- **Page Header:** Title ("Student Dashboard") and subtitle.
- **Flash Messages:** Area for displaying Flask flash messages.
- **Student Statistics Widgets (student-stats-widgets):** Four prominent cards displaying key statistics:
  - Total Wins
  - Total Tournaments Participated
  - Win Rate (%)
  - Number of Unique Sports Played
- **Student Profile Section (profile-section):** A card-modern displaying detailed student information (Name, USN, Branch, Section, Email, Phone).
  - Includes an optional "Agent Information" sub-section that appears only if agent details are available.
- **Tournament Entries Section (tournaments-section):**

- A header with a title ("Your Tournament Entries") and a button to "Add Tournament".
- **Tournament Grid (tournaments-grid):** If tournaments data is available, it iterates through each tournament and renders a tournament-card.
  - Each tournament-card displays: Tournament Name, Sport Name (with an icon), Date, League/Level, Event Type, Team Name/Role (if applicable), Opponents, Scores, Outcome (with a colored badge and icon), and Achievements/Awards (if any).
  - Action buttons for "Edit Tournament" and "Delete Tournament".
- **Empty State (empty-state)::** If no tournament entries exist, a message prompts the user to add their first tournament.

- **JavaScript:**
  - Includes Bootstrap's JavaScript bundle.
  - **Theme Switcher Logic:** Same as other pages.
  - **Alert Dismissal:** Adds functionality to close flash messages with a fade-out animation.
  - **Card Fade-in Animation:** Uses an IntersectionObserver to trigger a fade-in-up animation when tournament cards become visible in the viewport, creating a smooth loading effect.
  - **deleteTournament(tournamentId) function:**
    - Prompts for user confirmation before deletion.
    - Sends an AJAX POST request to /delete_tournament/<tournamentId>.
    - Handles success/error responses, displays custom notifications, and removes the deleted card from the DOM with an animation.
    - Includes a loading state for the delete button.
  - **showNotification(message, type) function:** A custom client-side function to display dynamic alert messages (used after AJAX operations), mimicking Flask's flash messages.
  - **createAlertsContainer() function:** Helper to ensure the alerts container exists in the DOM.

- **Global Error Handling:** Catches unhandled promise rejections (e.g., failed network requests) and displays a generic error notification.

6.6. add_edit_tournament.html

This HTML file serves as a dual-purpose form for both adding new tournament entries and editing existing ones. Its content dynamically changes based on the is_edit variable passed from Flask.

Key Components:

- **HTML Structure:** Standard boilerplate, similar to other pages, with links to necessary CSS and icon libraries.
- **Navigation Bar:** Consistent navbar with brand, a "Back to Dashboard" button, theme switcher, and logout.
- **Page Header:** Dynamically displays "Add New Tournament Entry" or "Edit Tournament Entry" with corresponding icons and subtitles.
- **Flash Messages:** Area for displaying Flask flash messages.
- **Tournament Form (card-modern):** The main form for entering/updating tournament details:
  - **Hidden sport_id input:** Stores the ID of the selected sport, which is populated by JavaScript.
  - **Sport Selection Grid (sport-selection-grid):** A visual grid of buttons for selecting a sport. Each button represents a sport and has a corresponding icon (Material Symbols or Font Awesome). The currently selected sport is highlighted.
    - Includes client-side validation to ensure a sport is selected.
  - Fields for:
    - Tournament Name
    - Date (HTML5 date input)
    - League/Level (dropdown)
    - Type of Event (dropdown: Singles, Doubles, Team Event, Individual)
    - **Dynamic Team Fields:** Team Name and Role / Primary Position fields that are shown and made required only when EventType is "Doubles" or "Team Event` .

        - Opponent(s)
        - Coach Name (optional)
        - Your Score (optional)
        - Opponent Score (optional)
        - Outcome (dropdown: Win, Loss, Draw, Participated)
        - Achievements or Award (optional)
        - Notes (textarea)
    - **Action Buttons:** Dynamically displays "Add Tournament" or "Update Tournament" and a "Cancel" button.
- **JavaScript:**
    - Includes Bootstrap's JavaScript bundle.
    - **Theme Switcher Logic:** Same as other pages.
    - **Sport Selection Logic:**
        - Handles clicks on sport icon buttons: removes selected class from others, adds to the clicked one, and updates the hidden selected_sport_id input.
        - Performs client-side validation on form submission to ensure a sport is selected.
    - **Team Fields Toggle Logic:**
        - Listens for changes on the event_type dropdown.
        - The toggleTeamFields() function dynamically sets the display style of the team_fields row to flex or none and adds/removes the required attribute and required star (*) for team_name and team_role inputs based on the selected event type.

6.7. admin_login.html

This HTML file provides the user interface for the administrator login page.

Key Components:

- **HTML Structure:** Standard HTML5 boilerplate with links to Bootstrap CSS, custom style.css, and Google Material Symbols Outlined.

- **Navigation Bar:** A simple navbar with the application brand and a theme switcher button.
- **Login Card (auth-card):** A centrally aligned card specifically for admin login:
    - A header with an admin icon, title ("Admin Access"), and subtitle.
    - A section for displaying Flask flash messages.
    - An admin login form with fields for Username and Password.
    - A "Sign In as Admin" button.
    - A "Back to Student Login" link.
- **JavaScript:**
    - Includes Bootstrap's JavaScript bundle.
    - **Theme Switcher Logic:** Handles saving and applying the user's preferred theme (light/dark) to localStorage.
    - **Input Focus Transitions:** Adds and removes a focused class to parent elements of form inputs when they gain/lose focus, enabling smooth visual effects defined in style.css.

6.8. set_admin_credentials.html

This HTML file is used for the initial setup of the administrator's username and password. It's designed to be shown only when no admin account exists in the database.

Key Components:

- **HTML Structure:** Standard HTML5 boilerplate with links to Bootstrap CSS, custom style.css, and Google Material Symbols Outlined.
- **Navigation Bar:** A simple navbar with a "Sport Portfolio Setup" brand and a theme switcher.
- **Setup Card (auth-card):** A centrally aligned card for admin setup:
    - A header with a security icon, title ("Admin Setup"), and subtitle.
    - A section for displaying Flask flash messages.
    - A form for creating admin credentials with fields for Admin Username, Password, and Confirm Password.

    - A "Set Credentials" button.
- **JavaScript:**
    - Includes Bootstrap's JavaScript bundle.
    - **Theme Switcher Logic:** Handles saving and applying the user's preferred theme (light/dark) to localStorage.
    - **Input Focus Transitions:** Adds and removes a focused class to parent elements of form inputs when they gain/lose focus.
    - **Password Confirmation Validation:** Client-side JavaScript that checks if the password and confirm_password fields match. It sets a custom validation message if they don't, preventing form submission.

6.9. admin_dashboard.html

This HTML file provides the user interface for the administrator's dashboard, offering an overview of the application's data.

Key Components:

- **HTML Structure:** Standard HTML5 boilerplate with links to Bootstrap CSS, style.css, Material Symbols, Font Awesome, and includes the Chart.js library.
- **Navigation Bar:** Includes the application brand, a welcome message for the admin, a theme switcher, and an admin logout button.
- **Page Header:** Title ("Admin Dashboard") and subtitle.
- **Flash Messages:** Area for displaying Flask flash messages.
- **Dashboard Widgets (dashboard-widgets):**
    - **Total Students Widget:** Displays the total count of registered students.
    - **Students by Sport Chart:** A canvas element that will render a pie chart using Chart.js, showing the distribution of students across different sports.
    - **Tournaments per Sport List:** A list displaying the count of tournaments recorded for each sport.
- **All Student Profiles Section:**
    - A header with a title ("All Student Profiles") and subtitle.
    - **Search and Filter Section:**

- A search input field (student-search-input) for filtering students by name, USN, branch, or sport.
- A "Clear Search" button.
- A set of "Sort Buttons" that allow sorting the student list by various criteria (Wins, Losses, Matches, Last Updated, Name, USN, Branch) in ascending or descending order.

- **Student Cards Container (student-cards-container):** A grid (student-grid) where individual student profile cards are dynamically rendered by JavaScript.
- **Empty State:** Displays a message if no student data is available or no students match the search criteria.

- **JavaScript:**

- Includes Bootstrap's JavaScript bundle.
- **Theme Switcher Logic:** Same as other pages, but also includes logic to update Chart.js colors when the theme changes.
- **Data from Flask:** Receives studentsData, studentsPerSportData, and tournamentsPerSportData as JSON objects from the Flask backend.
- **renderStudentCards(studentsToRender) function:**

- Takes an array of student objects and dynamically generates HTML cards for each student.
- Populates card details with student information, including calculated stats (wins, losses, matches, sports played).
- Applies a fade-in-up animation to each card.
- Displays an "No Students Found" empty state if the filtered list is empty.

- **filterAndSortStudents() function:**

- Filters studentsData based on the currentSearchQuery.
- Sorts the filtered list based on currentSortBy and currentOrder (handling both numeric, date, and string comparisons).
- Calls renderStudentCards() to update the displayed list.

- **Event Listeners:**

        - student-search-input (keyup): Triggers filterAndSortStudents on every key press.
        - clear-search (click): Clears the search input and re-filters.
        - sort-buttons (click): Updates currentSortBy and currentOrder, changes the sort icon, and triggers filterAndSortStudents.
    - **Chart.js Implementation:**
        - Initializes a pie chart using the sports-pie-chart canvas.
        - Uses studentsPerSportData to populate the chart labels (sport names) and data (student counts).
        - Defines a set of backgroundColor and borderColor for the chart segments.
        - Configures chart options for responsiveness, legend position, and tooltip callbacks to display student counts. The legend and tooltip colors are dynamically updated with the theme.

7. Database Management System (DBMS) - SQLite3

The application uses SQLite3, a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. It's widely used for embedded databases and local storage due to its serverless architecture (it doesn't require a separate server process).

7.1. Database Schema

The student_sport_portfolio.db database contains the following tables:

- **Students Table:**
    - StudentID (INTEGER PRIMARY KEY AUTOINCREMENT): Unique identifier for each student.
    - USN (TEXT UNIQUE NOT NULL): Unique Student Number (e.g., 1RV20CS001).
    - FirstName (TEXT NOT NULL)
    - LastName (TEXT NOT NULL)
    - DateOfBirth (DATE NOT NULL): Used as the student's password.
    - Email (TEXT NOT NULL)
    - CollegeJoiningYear (INTEGER)

- CollegeEndingYear (INTEGER)
- Branch (TEXT)
- Section (TEXT)
- PhoneNumber (TEXT)
- AgentName (TEXT)
- AgentPhoneNumber (TEXT)
- CreatedAt (TIMESTAMP DEFAULT CURRENT_TIMESTAMP): Timestamp of record creation.
- UpdatedAt (TIMESTAMP DEFAULT CURRENT_TIMESTAMP): Timestamp of last record update.

- **Sports Table:**
  - SportID (INTEGER PRIMARY KEY AUTOINCREMENT): Unique identifier for each sport.
  - SportName (TEXT UNIQUE NOT NULL): Name of the sport (e.g., 'Cricket', 'Football').
- **Tournaments Table:**
  - TournamentID (INTEGER PRIMARY KEY AUTOINCREMENT): Unique identifier for each tournament entry.
  - StudentID (INTEGER): Foreign key referencing Students.StudentID.
  - SportID (INTEGER): Foreign key referencing Sports.SportID.
  - TournamentName (TEXT NOT NULL)
  - TournamentDate (DATE NOT NULL)
  - LeagueLevel (TEXT NOT NULL): E.g., 'District', 'State', 'National'.
  - EventType (TEXT NOT NULL): E.g., 'Singles', 'Doubles', 'Team Event'.
  - TeamName (TEXT): Optional, for doubles/team events.
  - TeamRole (TEXT): Optional, for team events (e.g., 'Captain', 'Striker').
  - Opponents (TEXT NOT NULL)
  - CoachName (TEXT)

- StudentScore (TEXT)
- OpponentScore (TEXT)
- Outcome (TEXT NOT NULL): E.g., 'Win', 'Loss', 'Draw', 'Participated'.
- AchievementsAwards (TEXT): Optional.
- Notes (TEXT): Optional.
- CreatedAt (TIMESTAMP DEFAULT CURRENT_TIMESTAMP)
- UpdatedAt (TIMESTAMP DEFAULT CURRENT_TIMESTAMP)

- **Admin Table:**
  - AdminID (INTEGER PRIMARY KEY AUTOINCREMENT): Unique identifier for the admin.
  - Username (TEXT UNIQUE NOT NULL)
  - PasswordHash (TEXT NOT NULL): Hashed password for security.
  - CreatedAt (TIMESTAMP DEFAULT CURRENT_TIMESTAMP)

7.2. Database Operations

The app.py file interacts with the SQLite database using the sqlite3 module.

- Connecting to the Database:
  The get_db_connection() function establishes a connection to student_sport_portfolio.db. It's crucial that conn.row_factory = sqlite3.Row is set, as this allows fetching rows as dictionary-like objects, making column access by name (e.g., row['USN']) much more readable and maintainable than by index.

```
import sqlite3

def get_db_connection():
  conn = sqlite3.connect('student_sport_portfolio.db')
  conn.row_factory = sqlite3.Row # Enables dictionary-like access to rows
  return conn
```

- Creating Tables and Initial Data (init_db()):
  This function is called once when the application starts. It creates all necessary

tables (Students, Sports, Tournaments, Admin) if they don't already exist. It also populates the Sports table with a predefined list of sports, using INSERT OR IGNORE to ensure that these sports are only added once.

```
def init_db():
   conn = sqlite3.connect('student_sport_portfolio.db')
   cursor = conn.cursor()
   # ... (SQL CREATE TABLE statements for Students, Sports, Tournaments, Admin) ...
   # ... (SQL INSERT OR IGNORE statements for default Sports) ...
   conn.commit() # Saves all changes to the database
   conn.close() # Closes the connection
```

- Inserting Data:
  New student registrations and tournament entries are inserted using INSERT INTO SQL statements. Parameters are passed as a tuple to the execute method, preventing SQL injection vulnerabilities.
  Example: Student Signup

```
# In signup() route
conn.execute('''
   INSERT INTO Students (
      USN, FirstName, LastName, DateOfBirth, Email,
      CollegeJoiningYear, CollegeEndingYear, Branch, Section,
      PhoneNumber, AgentName, AgentPhoneNumber
   ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
''', (
   usn, first_name, last_name, dob, email,
   college_joining_year, college_ending_year, branch, section,
   phone_number, agent_name, agent_phone_number
))
conn.commit() # Commit the transaction to save changes
```

- Querying Data (SELECT):
  Data retrieval is performed using SELECT statements. The fetchone() method retrieves a single row, while fetchall() retrieves all matching rows.

Example: Student Login Check

```
# In login() route
student = conn.execute(
  'SELECT * FROM Students WHERE USN = ? AND DateOfBirth = ?',
  (usn, dob)
).fetchone() # Fetches one matching student record
```

Example: Fetching Tournaments for Dashboard

```
# In dashboard() route
tournaments = conn.execute('''
  SELECT t.*, s.SportName
  FROM Tournaments t
  JOIN Sports s ON t.SportID = s.SportID
  WHERE t.StudentID = ?
  ORDER BY t.TournamentDate DESC
''', (student_id,)).fetchall() # Fetches all tournaments for a student
```

- Updating Data:
  Existing records are modified using UPDATE SQL statements.
  Example: Editing a Tournament

```
# In edit_tournament() route
conn.execute('''
  UPDATE Tournaments SET
    SportID = ?, TournamentName = ?, TournamentDate = ?, LeagueLevel = ?,
    EventType = ?, TeamName = ?, TeamRole = ?, Opponents = ?, CoachName = ?,
    StudentScore = ?, OpponentScore = ?, Outcome = ?, AchievementsAwards = ?,
    Notes = ?, UpdatedAt = CURRENT_TIMESTAMP
  WHERE TournamentID = ? AND StudentID = ?
''', (
  sport_id, tournament_name, tournament_date, league_level,
  event_type, team_name, team_role, opponents, coach_name,
  student_score, opponent_score, outcome, achievements_awards, notes,
  tournament_id, student_id
))
```

```
conn.commit()
```

- Deleting Data:
  Records are removed using DELETE FROM SQL statements.
  Example: Deleting a Tournament
  ```
  # In delete_tournament() route
  conn.execute(
    'DELETE FROM Tournaments WHERE TournamentID = ? AND StudentID = ?',
    (tournament_id, student_id)
  )
  conn.commit()
  ```

- Transaction Management:
  The conn.commit() method is called after any INSERT, UPDATE, or DELETE operation to save the changes permanently to the database file. Without commit(), changes would not be persisted. conn.close() is called to release the database connection after operations are complete.

8. Usage Guide

8.1. Admin Setup and Login

1. **First Run:** When you run the application for the very first time, navigate to http://127.0.0.1:5000/admin. Since no admin credentials exist, you will be redirected to the "Admin Setup" page (/set_admin_credentials).
2. **Set Credentials:** Enter a desired Admin Username, Password, and Confirm Password. The password must be at least 6 characters long.
3. **Login:** After setting credentials, you will be redirected to the "Admin Login" page (/admin_login). Use the username and password you just created to log in.
4. **Subsequent Logins:** For all future admin logins, go to http://127.00.1:5000/admin_login.

8.2. Student Registration and Login

1. **Registration:** From the login page (http://127.0.0.1:5000/login), click on "New Student? Create Account".
2. **Fill Details:** Complete the registration form with your personal and academic details. Your Date of Birth will serve as your password. You can optionally add agent details.
3. **Login:** After successful registration, you will be redirected to the login page. Use your USN and Date of Birth to sign in.

8.3. Student Dashboard

Upon successful student login, you will be directed to your personalized dashboard:

- **Statistics Overview:** View your total wins, tournaments participated, win rate, and the number of unique sports you've played.
- **Profile Information:** See your registered details.
- **Tournament Entries:** A list of all tournaments you have added. Each card provides a summary and actions to edit or delete the entry.
- **Add Tournament:** Click the "Add Tournament" button to record a new sports achievement.

8.4. Adding/Editing Tournament Entries

1. **Adding:** From the student dashboard, click "Add Tournament".
2. **Editing:** On the student dashboard, click the "Edit" icon (pen icon) on any tournament card.
3. **Form Fields:**

- **Sport:** Select the sport from the interactive icon grid. This field is required.
- **Tournament Name, Date, League/Level, Type of Event, Opponent(s), Outcome:** These are required fields.
- **Team Name / Role:** These fields will dynamically appear and become required if you select "Doubles" or "Team Event" as the Type of Event.
- **Coach Name, Your Score, Opponent Score, Achievements or Award, Notes:** These fields are optional.

4. **Save:** Click "Add Tournament" (or "Update Tournament" if editing) to save your entry.

8.5. Admin Dashboard

After logging in as an administrator, you can access the admin dashboard:

- **Total Students:** A quick count of all registered students.
- **Students by Sport:** A pie chart showing the distribution of students across different sports they have participated in.
- **Tournaments per Sport:** A list indicating how many tournaments have been recorded for each sport.
- **All Student Profiles:** A comprehensive list of all registered students.
  - **Search:** Use the search bar to filter students by Name, USN, Branch, or Sport.
  - **Sort:** Use the sort buttons to arrange the student list by Wins, Losses, Matches, Last Updated, Name, USN, or Branch in ascending or descending order.

9. Conclusion

The Student Sport Portfolio application provides a robust and user-friendly platform for students to manage their athletic journey and for administrators to oversee student sports activities. Built with Flask and SQLite3, it offers a simple yet effective solution for tracking achievements, visualizing data, and maintaining student records. The modular design, clear file explanations, and detailed usage guide aim to facilitate easy understanding, maintenance, and potential future enhancements.