Submission Worksheet

CLICK TO GRADE

https://learn.ethereallab.app/assignment/IT202-452-M2024/it202-api-project-milestone-2-2024-m24/grade/vs53

IT202-452-M2024 - [IT202] API Project Milestone 2 2024 M24

Submissions:

Submission Selection

1 Submission [active] 7/22/2024 1:58:57 AM

Instructions

COLLAPSE

Implement the Milestone 2 features from the project's proposal document: https://docs.google.com/document/d/1XE96a8DQ52Vp49XACBDTNCq0xYDt3kF29cO88EWVwfo/view Make sure you add your ucid/date as code comments where code changes are done All code changes should reach the Milestone2 branch Create a pull request from Milestone2 to dev and keep it open until you get the output PDF from this assignment. Gather the evidence of feature completion based on the below tasks. Once finished, get the output PDF and copy/move it to your repository folder on your local machine. Run the necessary git add, commit, and push steps to move it to GitHub Complete the pull request that was opened earlier Create and merge a pull request from dev to prod Upload the same output PDF to Canvas

Branch name: Milestone2

Tasks: 23 Points: 10.00

Define Appropriate Tables for Data (1 pt.)

COLLAPSE

Task #1 - Points: 1

Text: Screenshots of Table SQL

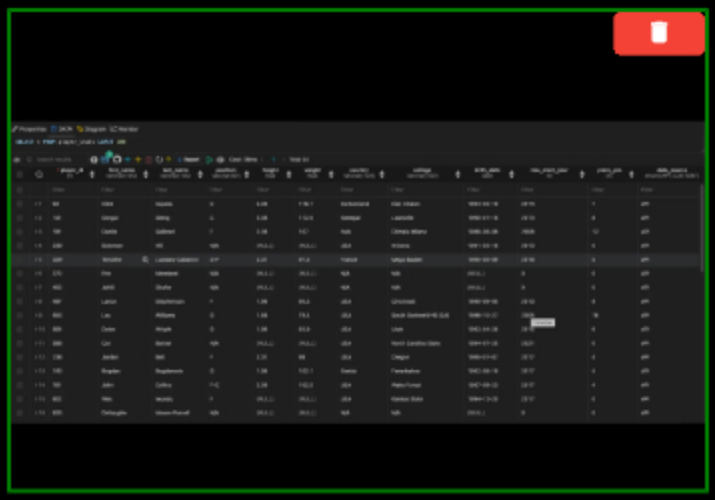
COLLAPSE

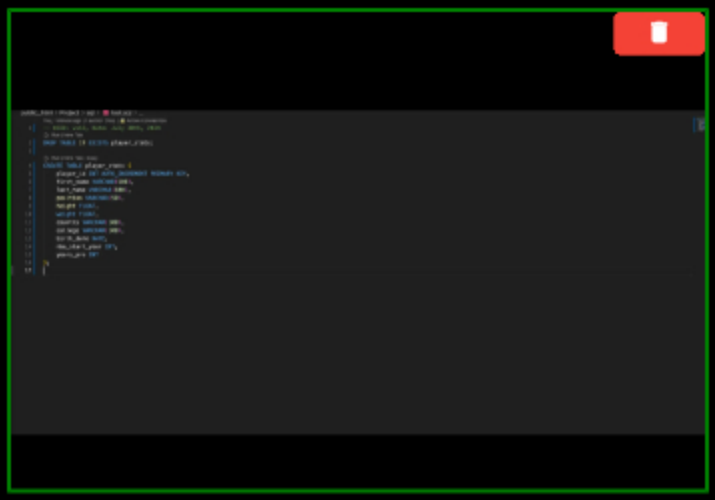
Checklist *The checkboxes are for your own tracking

#	Points	Details

#1	1	Table(s) should have the 3 core columns we'll always be using (id, created, modified) plus additional columns for the incoming API data
#2	1	Columns should be logical and thought out (not valid to have a single field of JSON data or similar

#1) Screenshot of the table (you can duplicate this subtask as needed)





Caption (required) ✓

Describe/highlight what's being shown

Two screenshots showing the table and the sql query to make the table

Explanation (required) ✓

Explain the columns and what data they represent (briefly), also note any normalization that may have been necessary

PREVIEW RESPONSE

In the player_stats table, each column represents a specific piece of information about basketball players. The player_id column is an auto-incremented primary key that uniquely identifies each player. The first_name and last_name columns store the player's first and last names. The position column indicates the player's position on the team, like guard or forward. The height and weight columns capture the player's physical stats, with height in meters and weight in kilograms. The country column shows the player's country of origin, while college notes where they played college basketball if applicable. The birth_date column records their date of birth, and the nba_start_year column notes the year they started in the NBA. And one for API to denote whether the data was custom made or from the API I was using. Finally, the years_pro column indicates how many years they've been playing professionally. The table doesn't seem to need normalization right now, as all the data is directly related to each player and fits well in a single table.

COLLAPSE

Task #2 - Points: 1

Text: Add the pull request link for the branch related to this feature

Details:

Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature.

URL #1
https://github.com/VikShah/vs53-it202-452/pull/44

URL
https://github.com/VikShah/vs53-it202-452/pull/

+ ADD ANOTHER URL

Data Creation Page (2 pts.)

COLLAPSE

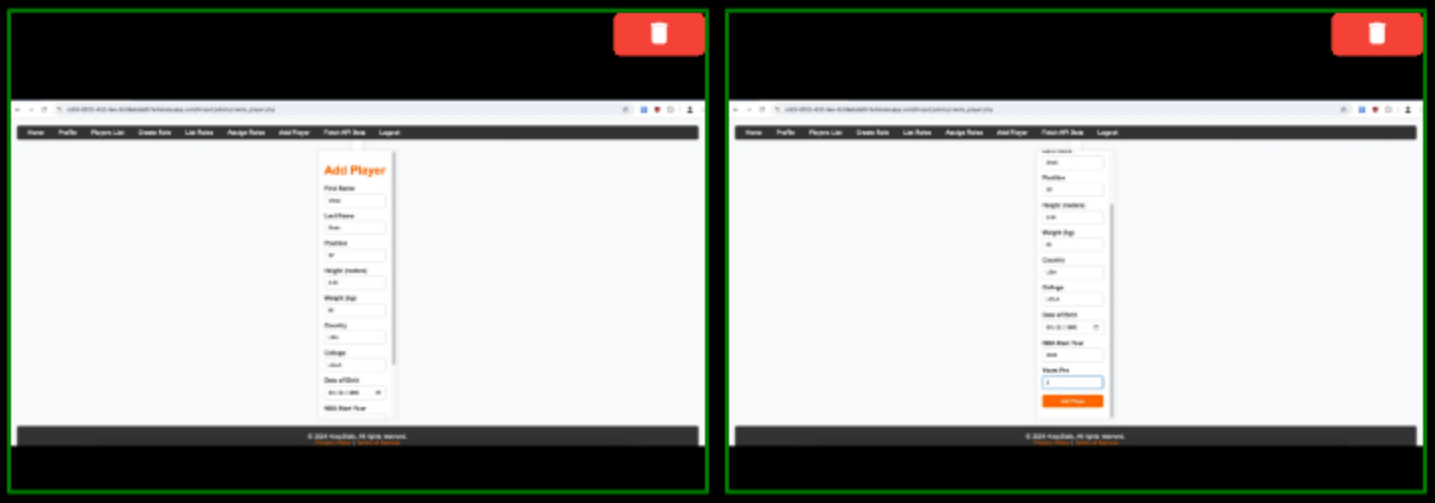
COLLAPSE

Task #1 - Points: 1

Text: Screenshots of the creation page

Details:
Heroku dev url must be visible in all relevant screenshots

#1) Show potentially valid data filled in for the custom creation page



Caption (required) ✓
Describe/highlight what's being shown
PIcture of the data creating page, where you add a player is the data creation

#2) Show how the API data is fetched for API data (must be server-side)

Caption (required) ✓

Describe/highlight what's being shown

Picture of the code with my UCID and date showing how I fetcehd teh API and key

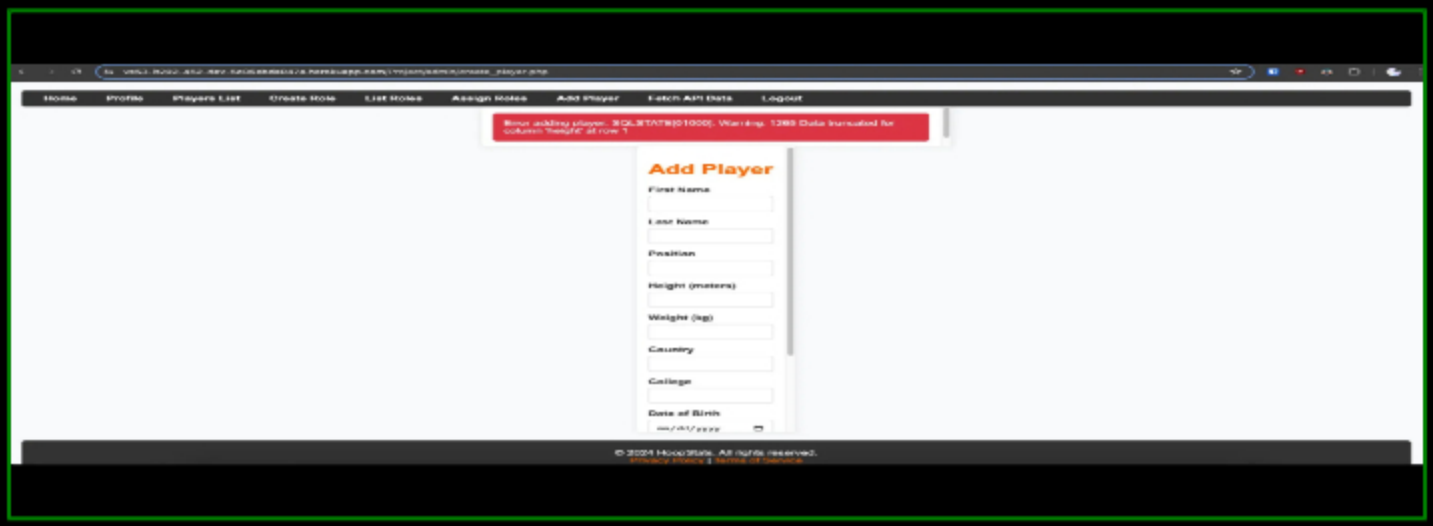
Explanation (required) ✓

Briefly explain the code

PREVIEW RESPONSE

In the fetch_api_data.php file, I wrote code to fetch player data from an NBA API and store it in a local database. I started by setting up error reporting to help catch any issues. Then, I included a functions file, which I use for database connections and other utility functions. In the main part of the script, I defined a function called fetchAndStoreApiData() that handles everything. I use my API key to access player data from the NBA API and make a request using cURL. After getting the data, I clear out the old data from the player_stats table in my database and insert the new data. The script ensures that if a player already exists in the database, their information gets updated rather than duplicated. Finally, I check the response to make sure the data is structured as expected.

#3) Show examples of validation messages

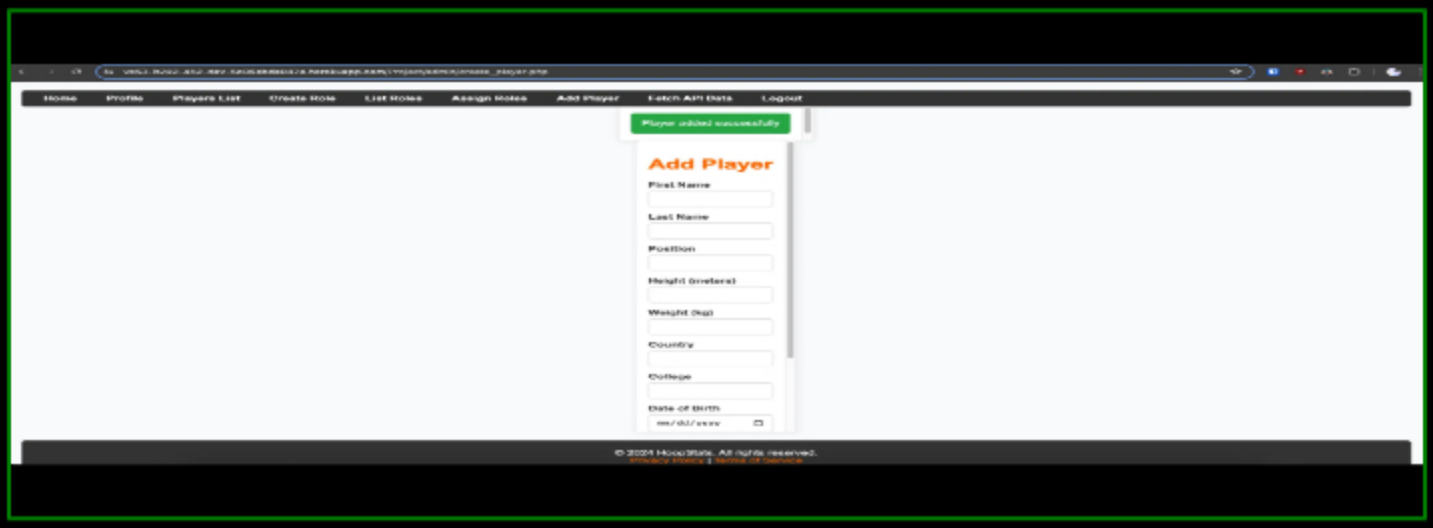


Caption (required) ✓

Describe/highlight what's being shown

An example of an valdiation message for putting incorrect parameters for height

#4) Show an example of successful creation message



Caption (required) ✓

Describe/highlight what's being shown

An example above if you create the player and the message you get

#5) Design/Style should be considered (i.e., bootstrap, custom css, etc)

Caption (required) ✓

Describe/highlight what's being shown

A picture of the design choice for the add player/data creation page

Explanation (required) ✓

Briefly explain your design choices

PREVIEW RESPONSE

For the design choices, I aimed for a clean and straightforward layout that complements the basketball theme of the website. I chose a black and orange color scheme, reflecting the colors often associated with basketball, like a basketball and the court. The design includes easy-to-navigate menus and clearly labeled sections, ensuring users can find information about players and their stats quickly. The layout uses responsive design principles, making it accessible on both desktops and mobile devices. The use of tables and structured forms helps in organizing the

data neatly, providing a user-friendly experience. I also added a footer section to provide a consistent end to the pages and offer additional navigation links or information. The overall goal was to create an interface that is both visually appealing and functional, enhancing the user experience while maintaining a sporty, energetic vibe.

COLLAPSE

Task #2 - Points: 1

Text: Screenshots of creation page code

Details:
Include ucid/date comments for each code screenshot

#1) Form should have correct data types for each property being requested

Caption (required) ✓

Describe/highlight what's being shown

Above is the screenshot of the creation page

Explanation (required) ✓

Briefly talk about each field type and why it was chosen

PREVIEW RESPONSE

In the create_player.php form, I used different field types to ensure the data we collect is accurate. For the player's first name, last name, and position, I used type="text" because these are simple text fields and can include letters, spaces, and sometimes other characters. For height and weight, even though they're numbers, I went with text fields to handle cases like decimal points or different units of measurement. The country and college fields are also text since they are names and can contain a variety of characters. I chose type="date" for the birth date to ensure we get the correct format and valid dates. Lastly, for the NBA start year and years pro, I stuck with text fields for flexibility, even though these are typically numbers. This setup helps keep our data consistent and avoids input errors.

#2) Form should have correct validation for each field (HTML, JS, and PHP)

Caption (required) ✓

Describe/highlight what's being shown

3 screenshots showing all the valdations, PHP HTML And JS

Explanation (required) ✓

Briefly explain the validations

PREVIEW RESPONSE

In this code, I implemented validations to ensure data integrity when adding a player. PHP validation checks server-side that all required fields are filled and values are appropriate, like making sure height and weight are positive numbers. This helps prevent bad data from entering the database. On the client side, I used JavaScript to give users immediate feedback if they try to submit the form with missing or incorrect information, like empty fields or non-numeric input for numbers. HTML attributes like required and type enforce basic validation rules directly in the form, making sure users fill in all necessary fields and use the correct data types. These validations work together to make the data entry process smoother and more reliable.

#3) Successful creation should have a user-friendly message

Caption (required) ✓

Describe/highlight what's being shown

Showing the underlined part where the succesful creation with flash a user-friendly message

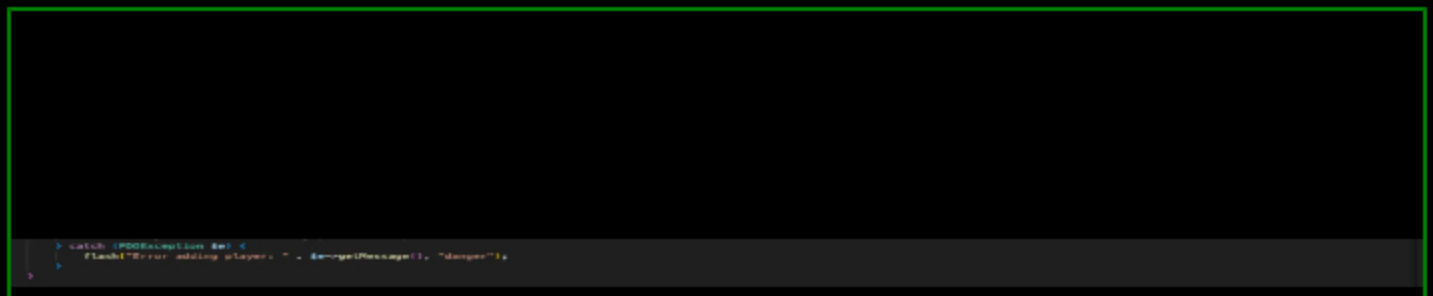
Explanation (required) ✓

Explain how duplicate/existing data is handled

PREVIEW RESPONSE

I handle duplicate or existing data using the ON DUPLICATE KEY UPDATE clause in my SQL query in my fetch_api_data.php file. When I fetch player data from the API and try to insert it into the database, this clause checks if a player with the same player_id already exists. If it does, instead of creating a new entry, it updates the existing record with the new data. This way, I prevent duplicates and ensure that the player information is always current. This method keeps my database clean and accurate, which is essential for my project. Using this approach helps me manage the data efficiently and avoids potential issues with duplicate entries.

#4) Any errors should have user-friendly messages



Caption (required) ✓

Describe/highlight what's being shown

Catching any errors and displaying the error meesage.

Explanation (required) ✓

Briefly describe the scenarios

PREVIEW RESPONSE

When adding a player, there are a few scenarios where user-friendly error messages come into play. If any required fields are left blank, the form won't submit due to HTML and JavaScript validations, prompting the user to fill in the missing information. On the server side, if there is an issue with the database, such as a duplicate entry or invalid data, the PHP code catches this and uses the flash function to display a clear error message. For instance, if a player with the same ID already exists, or if the height is not a valid number, the user will see an appropriate error message. These messages help guide the user to correct the input and ensure that the data entered is accurate and complete.

#5) Include the form/process for fetching API data

Caption (required) ✓

Describe/highlight what's being shown

Showing the code for the process of fetching the API data

Explanation (required) ✓

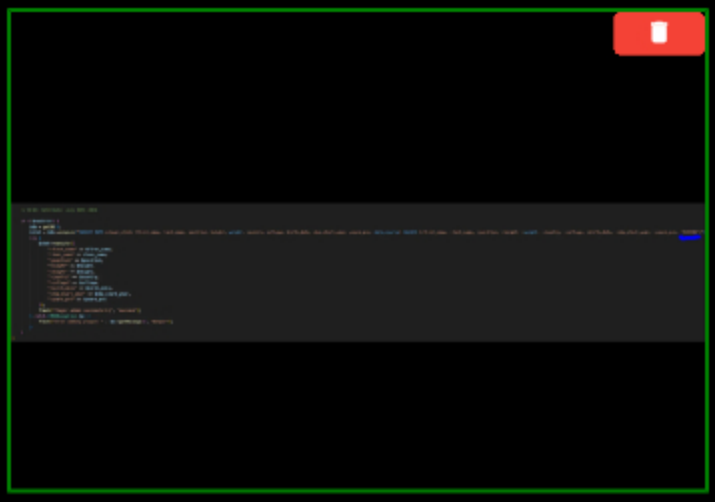
Briefly explain the steps (include how duplicates are handled)

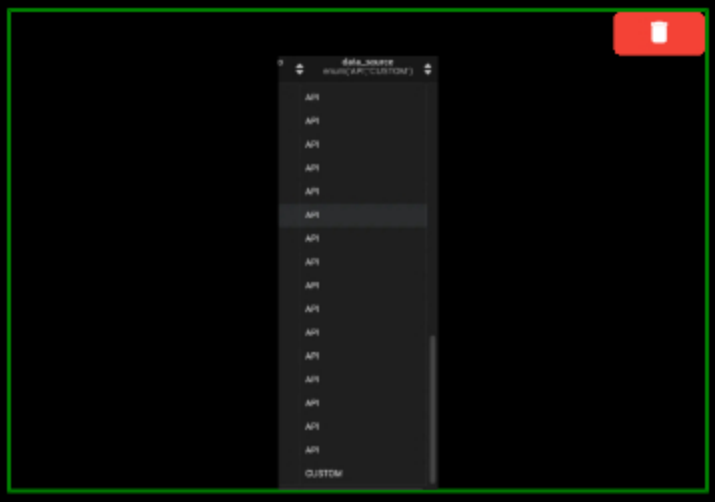
PREVIEW RESPONSE

In the fetch_api_data.php file, I set up error reporting to catch any issues while running the script. I included my functions file for database connections and utilities. Then, I defined a function called fetchAndStoreApiData() to

fetch player data from the NBA API using my API key. I used cURL to make the API request and decode the JSON response. Before inserting new data, I cleared out the existing player data from the database to avoid old data mixing with new data. Each player's data is then inserted into the player_stats table. If a player already exists, their information gets updated instead of duplicated. This way, I ensure that my database has the most recent data without any duplicates. Finally, I check the API response to make sure the data structure is as expected.

#6) Include some indicator between custom data and API data





Caption (required) ✓

Describe/highlight what's being shown

Two screenshots, one underlining how the data that I create is marked as "Custom" and a part of my database table showing that.

Explanation (required) ✓

Briefly mention what the indicator is (i.e., api_id if the API has ids or is_api as a boolean-like column, etc)

PREVIEW RESPONSE

In my project, I used a column called data_source as an indicator to differentiate between custom data and API data. When a player is manually added through the creation form, the data_source value is set to 'Custom'. For players fetched from the API, the data_source value is set to 'API'. This helps in identifying the origin of each player's data in the database. It makes it easier to manage and update the records accordingly.

#7) Include any other rules like role guards and login checks

```php
// UCID: Vasu Date: July 28th 2024

// the user is logged in and has admin privileges      You, 20 minutes ago • Uncommitted changes
is_logged_in(true);
if (!has_role("Admin")) {
    flash("You do not have permission to access this page", "danger");
    die(header("Location: " . get_url("home.php")));
}
```

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing that only admins can create players/add players

Explanation (required) ✓

Briefly explain the logic/reasoning

PREVIEW RESPONSE

In my project, I included login checks and role guards to ensure that only authorized users can perform certain actions. For instance, in create_player.php and fetch_api_data.php, I added checks to verify if a user is logged in and has the "Admin" role. This way, only admins can add new players or fetch API data, preventing unauthorized access and potential misuse of these features. The logic behind this is to maintain security and integrity within the application.

COLLAPSE

Task #3 - Points: 1

Text: Screenshot of records from DB

#1) Show at least one record fetched from the API

Caption (required) ✓

Describe/highlight what's being shown (note what differs from a custom record)

Showing NBA player Kevin Huerter, notice that the last column says "API" which differs it from a custom record

#2) Show at least one record created via the creation form

Caption (required) ✓

Describe/highlight what's being shown (note what differs from the API record)

Showing record from fake NBA player Vikas Shah (My legal name), notice that last column which says "CUSTOM" to differ from API

COLLAPSE

Task #4 - Points: 1

Text: Add related links

Checklist *The checkboxes are for your own tracking

#	Points	Details
#1	1	Include the heroku prod link for this page
#2	1	Add the pull request link for the branch related to this feature Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature

URL #1
https://github.com/VikShah/vs53-it202-452/pull/45

URL
https://github.com/VikShah/vs53-it202-452/pull

URL #2
https://vs53-it202-452-prod-1c77a1c304b2.herokuapp.com/Project/admin/create_player.php

URL
https://vs53-it202-452-prod-1c77a1c304b2.herol

+ ADD ANOTHER URL

Data List Page (many entities) (2 pts.)

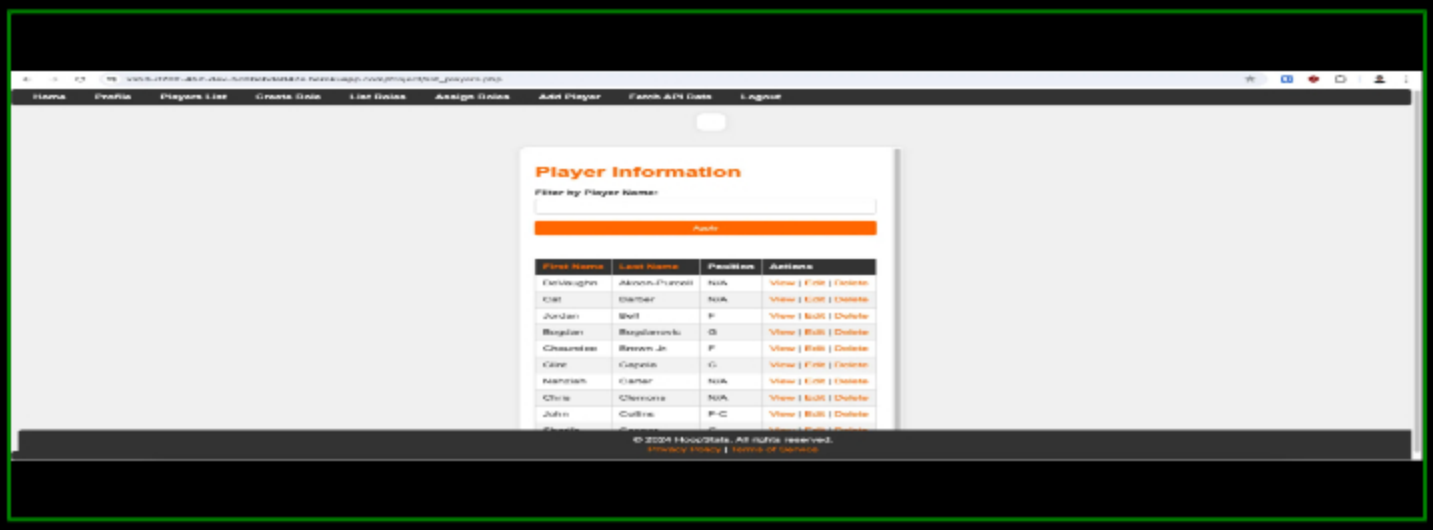
COLLAPSE

Task #1 - Points: 1

Text: Screenshots of the list page

Details:
Heroku dev url must be visible in all relevant screenshots

#1) Show the page of your entities listed (have a reasonable number shown)



Caption (required) ✓
Describe/highlight what's being shown
Screenshot of the players (entities) listed page

Explanation (required) ✓
Briefly describe the page
PREVIEW RESPONSE
The page I created lists all the players in the database, displaying their basic information like first name, last name, and position. I made sure to include a reasonable number of players per page to keep it manageable and easy to navigate. Each player entry includes links to view more details, edit the player's information, or delete the player if the user has admin privileges. The page also has filtering and sorting options to help users find specific players quickly. This setup makes it straightforward for users to browse through the list and manage player data efficiently.

#2) Show the filter/sort form based on your data and the required limit field

Caption (required) ✓

Describe/highlight what's being shown

Showing the filter/sort options and a screenshot of that, limit field of 100

Explanation (required) ✓

Briefly mention what's available to the user

PREVIEW RESPONSE

On the filter and sort form, users can search for players by name and specify how many players they want to see per page, with a limit between 1 and 100. They can also sort the list of players by first name or last name in either ascending or descending order. This makes it easy to quickly find and organize player information based on their preferences.

#3) Demonstrate a few varied filters/sorts

Caption (required) ✓

Describe/highlight what's being shown

Showing filtering by first name (John) and last name (Capela)

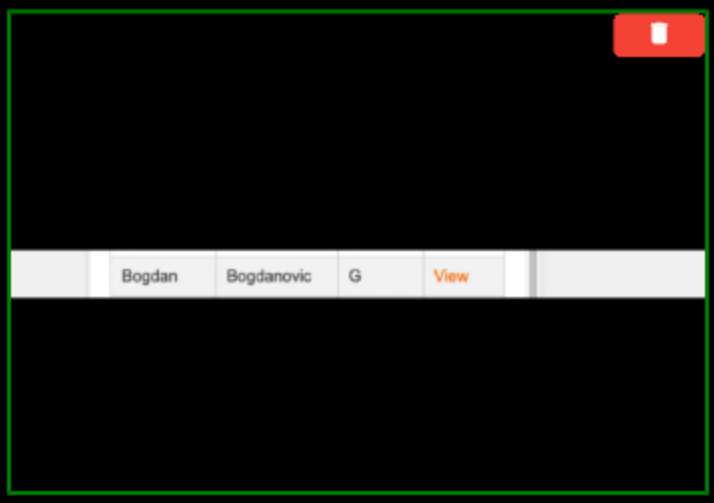
#4) Demonstrate a filter that doesn't have any records (should show an appropriate message)

Caption (required) ✓

Describe/highlight what's being shown

Showing the message you get when you demonstrate a filter without any records

#5) Each list item should have a link of single view (i.e., details), edit, and delete (some of which may only be visible to admin users, include examples from different user roles)



Caption (required) ✓

Describe/highlight what's being shown

Notice the difference between the admin point of view, and the user point of view. Showing 2 screenshots showing the difference.

Explanation (required) ✓

Mention which users can interact with the view, edit, and delete links

PREVIEW RESPONSE

On the player list page, I've set it up so that regular users can only view player details by clicking on the "View" link. Admins, however, have additional options. They can edit or delete player information using the "Edit" and "Delete" links. This distinction helps keep the data secure and manageable by limiting modification permissions to admins only while still allowing regular users to access the information they need.

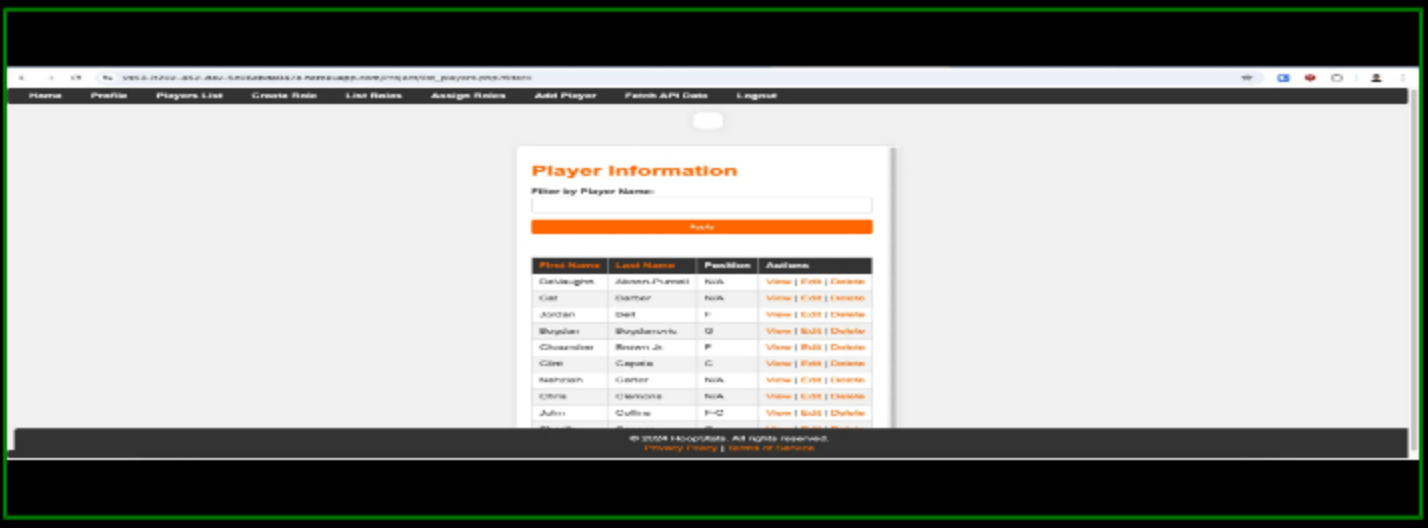
#6) Each list item should have a summary of the entity (likely won't be the entire entity data)

Caption (required) ✓

Describe/highlight what's being shown

Shows basic detail, the first name, the last name, and the NBA POSITION

#7) Design/Style should be considered (i.e., bootstrap, custom css, etc)



Caption (required) ✓

Describe/highlight what's being shown

Screenshot of the full view player list page

Explanation (required) ✓

Briefly explain your design choices

PREVIEW RESPONSE

For the design of my HoopStats site, I went with a clean, simple layout that fits the basketball theme. I used a black and orange color scheme to match basketball colors, like a basketball and the court. I made sure the navigation is easy to use, with clearly labeled sections for quick access to player stats and information. I used tables to organize data neatly and forms to make input straightforward. There's also a footer with extra navigation links to keep the layout consistent across pages. The overall goal was to make the site user-friendly and visually appealing, while keeping a sporty vibe.

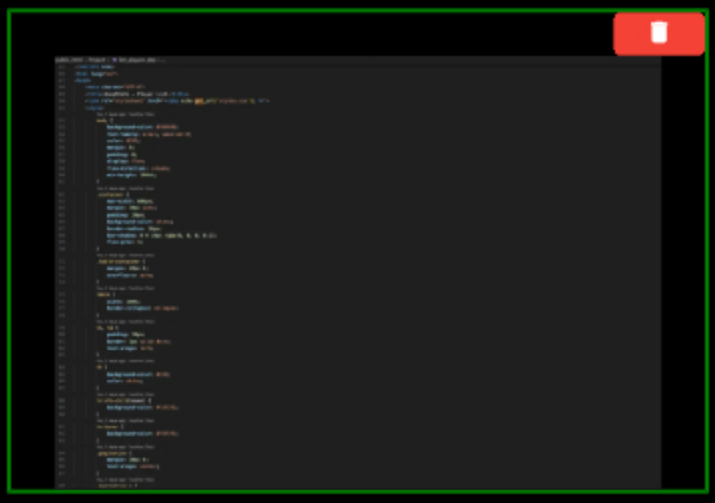
Task #2 - Points: 1

^COLLAPSE ^

Text: Screenshots of the list page code

Details:
Include ucid/date comments for each code screenshot

#1) Show the filter/sort form generation



Caption (required) ✓

Describe/highlight what's being shown

Two screenshots of the filter/sort form generations

Explanation (required) ✓

Briefly explain how the code works (i.e, some stuff may be dynamic)

PREVIEW RESPONSE

In this section of the code, I set up a form to filter player names. The form uses a GET method to send filter input. This allows users to type a player's name to filter the list. The PHP code handles the filter input and passes it to the database query.

#2) Show the DB query and how the filter/sort is handled (including the restriction on the limit field)

```php
// Fetch player stats with filtering, sorting, and pagination
// UCID: Vs53 Date: July 30th 2024
    You, days ago • Uncommitted changes
$db = getDB();
$query = "SELECT * FROM player_stats WHERE first_name LIKE :filter OR last_name LIKE :filter ORDER BY $sort $order LIMIT :limit OFFSET :offset";
$stmt = $db->prepare($query);
$filterParam = "%" . $filter . "%";
$stmt->bindParam(":filter", $filterParam, PDO::PARAM_STR);
$stmt->bindParam(":limit", $limit, PDO::PARAM_INT);
$stmt->bindParam(":offset", $offset, PDO::PARAM_INT);
$stmt->execute();
$players = $stmt->fetchAll(PDO::FETCH_ASSOC);

// Count total records for pagination
$totalQuery = "SELECT COUNT(*) FROM player_stats WHERE first_name LIKE :filter OR last_name LIKE :filter";
$totalStmt = $db->prepare($totalQuery);
$totalStmt->execute([":filter" => $filterParam]);
$totalRecords = $totalStmt->fetchColumn();
```

```
$totalPages = ceil($totalRecords / $limit);
```

Caption (required) ✓

Describe/highlight what's being shown

Screenshot of the DB query and how its handled

Explanation (required) ✓

Briefly explain the related logic

PREVIEW RESPONSE

In the list_players.php file, the database query is designed to fetch player stats with filtering, sorting, and pagination. The user can input a filter value to search by first or last name, and select sorting options for the columns. The limit field restricts the number of records displayed per page, and the server-side code ensures the limit is within the range of 1 to 100. The query uses bind parameters to safely include user inputs and avoid SQL injection. The results are fetched and displayed in a table, showing only a subset of data based on the filter and sort options chosen.

#3) Show how the output is generated and displayed

```
<div class="table-container">
    <table>
        <thead>
            <tr>
                <th><a href="?sort=first_name&order=<?php echo $order === 'ASC' ? 'DESC' : 'ASC'; ?>">First Name</a></th>
                <th><a href="?sort=last_name&order=<?php echo $order === 'ASC' ? 'DESC' : 'ASC'; ?>">Last Name</a></th>
                <th>Position</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <?php if (empty($players)) : ?>
                <tr>
                    <td colspan="4">No results available</td>
                </tr>
            <?php else : ?>
                <?php foreach ($players as $player) : ?>
                    <tr>
                        <td><?php se($player, "first_name"); ?></td>
                        <td><?php se($player, "last_name"); ?></td>
                        <td><?php se($player, "position"); ?></td>
                        <td>
                            <a href="view_player.php?id=<?php se($player, "player_id"); ?>">View</a>
                            <?php if ($isAdmin) : ?>
                                | <a href="<?php echo get_url('admin/edit_player.php?id=' . se($player, "player_id", "", false)); ?>">Edit</a>
                                | <a href="<?php echo get_url('admin/delete_player.php?id=' . se($player, "player_id", "", false)); ?>">Delete</a>
                            <?php endif; ?>
                        </td>
                    </tr>
                <?php endforeach; ?>
            <?php endif; ?>
        </tbody>
    </table>
</div>
```

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing how the output is generated

Explanation (required) ✓

Briefly explain the related logic

PREVIEW RESPONSE

In the list_players.php file, the output is generated by fetching player data from the database based on the filter, sort, and pagination settings. After executing the query, the results are stored in an array. The HTML table is then dynamically populated with these results. Each row in the table represents a player, displaying their first name, last name, and position. Additionally, there are action links for viewing, editing, and deleting players, which are conditionally shown based on the user's role. The table and pagination controls ensure the data is presented in a

structured and user-friendly manner.

#4) Show any restrictions like role guard or login checks

```php
<?php
require(__DIR__ . "/../../partials/nav.php");

// Check if the user is logged in
// UCID: Vs53 Date: July 30th 2024

if (!is_logged_in()) {
    flash("You must be logged in to view this page", "warning");
    die(header("Location: login.php"));
}

// Check if the user has admin role for admin-specific functionalities
$isAdmin = has_role("Admin");
```

Caption (required) ✓

Describe/highlight what's being shown

Sceenshot showing the restrictions like checking for admin

Explanation (required) ✓

Briefly explain the logic/reasoning

PREVIEW RESPONSE

In list_players.php, I included role guards and login checks to control access to certain functionalities. I first check if the user is logged in using is_logged_in(). If not, they are redirected to the login page with a warning message. For admin-specific actions like editing or deleting players, I use has_role("Admin") to verify if the logged-in user has admin privileges. This ensures that only authorized users can perform sensitive actions, maintaining the security and integrity of the application. This logic helps prevent unauthorized access and modifications to player data.

COLLAPSE

Task #3 - Points: 1

Text: Add related links

Checklist

*The checkboxes are for your own tracking

#	Points	Details
#1	1	Include the heroku prod link for this page
#2	1	Add the pull request link for the branch related to this feature Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature

URL #1

https://vs53-it202-452-prod-1c77a1c304b2.herokuapp.com/Project/list_players.php

URL

https://vs53-it202-452-prod-1c77a1c304b2.herol

URL #2
https://github.com/VikShah/vs53-it202-452/pull/46

URL
https://github.com/VikShah/vs53-it202-452/pull/

+ ADD ANOTHER URL

View Details Page (single entity) (1 pt.)

^COLLAPSE^

^COLLAPSE^

Task #1 - Points: 1

Text: Screenshots of the details page

Details:
Heroku dev url must be visible in all relevant screenshots

#1) Entity should be fetch by id (via the url)

Caption (required) ✓
Describe/highlight what's being shown
Screenshowing showing the fetch by id via the url

#2) A missing id should redirect back to the list page with an applicable message

Caption (required) ✓

Describe/highlight what's being shown

Showing what happens if you put an missing id, the message shows up

#3) Design/Style should be considered (i.e., bootstrap, custom css, etc)

Caption (required) ✓

Describe/highlight what's being shown

Showing the nice orange and black, sportish theme of the page.

Explanation (required) ✓

Briefly explain your design choices

PREVIEW RESPONSE

For the design choices of my project, I aimed for a clean and straightforward layout that complements the basketball theme of the website. I chose a black and orange color scheme to reflect the colors often associated with basketball, like a basketball and the court. I made sure the navigation bar was easy to use and consistent across all pages. The buttons and forms were styled to be user-friendly and visually appealing. I used custom CSS to enhance the overall look, making sure it was responsive and looked good on both desktop and mobile devices. For the flash messages, I chose different colors for success, warning, and danger messages to make them easily distinguishable. The table layout for listing players is simple yet effective, ensuring that the data is easy to read and interact with

#4) Data shown should be more detailed/inclusive than the summary view

Caption (required) ✓

Describe/highlight what's being shown

Comparing the two pages, showing how the data shown is more detailed/inclusive than the summary view.

Explanation (required) ✓

Briefly explain the details shown and how it differs from the list view

PREVIEW RESPONSE

In the detailed view, I included more comprehensive information about each player compared to the summary list view. The list view displays basic details like the player's first name, last name, and position. However, in the detailed view, I added additional information such as height, weight, years pro, college, country, date of birth, NBA start year, and position. This gives a fuller picture of the player's background and stats. The detailed view is designed to provide users with all the information they might need about a player in one place, making it more informative than the summary list view.

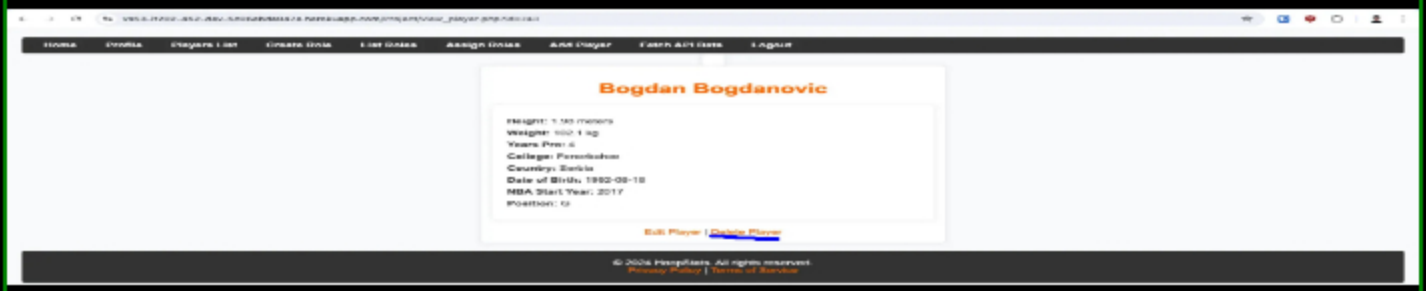
#5) There should be a link to edit the entity (this may be an admin-only thing, but it should be present for the respective role)

Caption (required) ✓

Describe/highlight what's being shown

Underlined in blue the edit entity button only for admins

#6) There should be a link to delete the entity (this may be an admin-only thing, but it should be present for the respective role)



Caption (required) ✓

Describe/highlight what's being shown

Underlined in blue the delete entity button only for admins

^COLLAPSE ^

Task #2 - Points: 1

Text: Screenshots of the details page code

Details:

Include ucid/date comments for each code screenshot

#1) Show how id is fetched

```
require(__DIR__ . "/../../lib/functions.php");
$player_id = $_GET['id'] ?? null;
// UCID: vs53 Date: July 30th, 2024
        You, last week • Added the API key and fetching ablitly
// Check if player_id is provided and is a valid integer
if (!$player_id || !filter_var($player_id, FILTER_VALIDATE_INT)) {
    flash("Invalid or missing player ID", "warning");
    header("Location: " . get_url("list_players.php"));
    exit;
}
```

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing how the id is fetched

Explanation (required) ✓

Briefly explain the logic and how it's handled when the property is missing or not "valid"

PREVIEW RESPONSE

In this part of the code, I fetch the player ID from the URL using $GET['id']. If the ID is not provided or is not a valid integer, I handle this by displaying a flash message saying "Invalid or missing player ID" and redirecting the user back to the list of players. This helps to prevent invalid or malicious inputs from being processed. By checking the validity of the player ID early on, I make sure the rest of the code only runs if the ID is correct. This way, if someone tries to access the details page with an invalid ID, they won't see an error page but will be redirected appropriately with a message.

#2) Show the DB query to get the record

```php
// UCID: vs53 Date: July 30th, 2024
      You, 8 minutes ago • Final changes
$db = getDB();
$stmt = $db->prepare("SELECT * FROM player_stats WHERE player_id = :player_id");
$stmt->execute([":player_id" => $player_id]);
$player = $stmt->fetch(PDO::FETCH_ASSOC);

// Check if player exists
if (!$player) {
    flash("Player not found", "danger");
    header("Location: " . get_url("list_players.php"));
    exit;
}
?>
```

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing the DB query to get the records

Explanation (required) ✓

Briefly explain the logic

PREVIEW RESPONSE

In this part of the code, I prepare and execute a SQL query to fetch the player's details from the player_stats table using the provided player ID. I use a prepared statement to avoid SQL injection and pass the player ID as a parameter to the query. The query selects all columns where the player_id matches the provided ID. If a matching record is found, it gets stored in the $player variable. This step is crucial to retrieve the specific player's information from the database to display on the details page.

#3) Show the code related to presenting the data and showing the links

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing the part relating to presenting the data and showing links

Explanation (required) ✓

Briefly explain the logic

PREVIEW RESPONSE

In this part of the code, I present the player's details on the webpage using HTML. The data fetched from the database is displayed in a structured format, such as the player's height, weight, years pro, college, country, date of birth, NBA start year, and position. I use PHP to insert the player's details into the HTML. I include links for editing and deleting the player. These links are only visible to admin users, using a conditional statement to check if the user has the admin role. This setup allows admins to manage player data directly from the details page.

COLLAPSE

Task #3 - Points: 1

Text: Add related links

Checklist

*The checkboxes are for your own tracking

#	Points	Details
☐ #1	1	Include the heroku prod link for this page
☐ #2	1	Add the pull request link for the branch related to this feature Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature

URL #1

https://vs53-it202-452-prod-1c77a1c304b2.herokuapp.com/Project/view_player.php?id=743

URL

https://vs53-it202-452-prod-1c77a1c304b2.herol

URL #2

https://github.com/VikShah/vs53-it202-452/pull/47

URL

https://github.com/VikShah/vs53-it202-452/pull,

+ ADD ANOTHER URL

Edit Data Page (2 pts.)

^COLLAPSE ^

^COLLAPSE ^

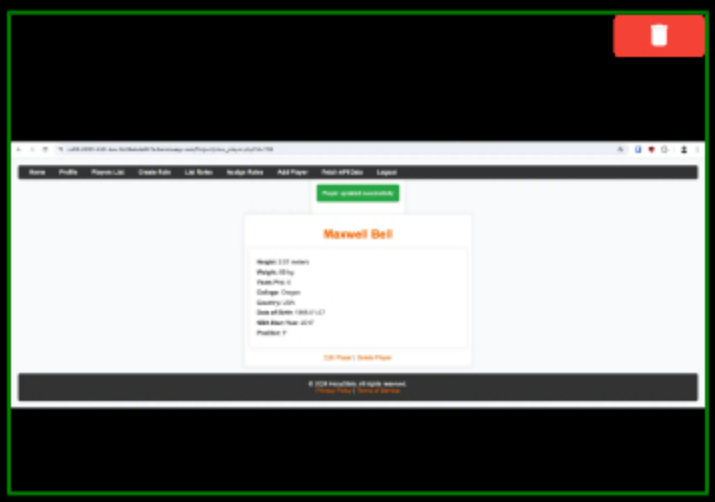
Task #1 - Points: 1

Text: Screenshots of the edit page

Details:

Heroku dev url must be visible in all relevant screenshots

#1) Show before and after screenshots of data you'll edit (two screenshots required)



Caption (required) ✓

Describe/highlight what's being shown

Showing 3 screenshots of the before, during, and after process of editing the data

Explanation (required) ✓

Explanation (required) ✓

Briefly explain what you changed

PREVIEW RESPONSE

I changed NBA player Jordan Bell's name from "Jordan Bell" to "Maxwell Bell"

#2) Show examples of validation messages

Caption (required) ✓

Describe/highlight what's being shown

Javascript red valdiation message not letting me update the height because of incorrect data type

#3) Show an example of successful edit messages

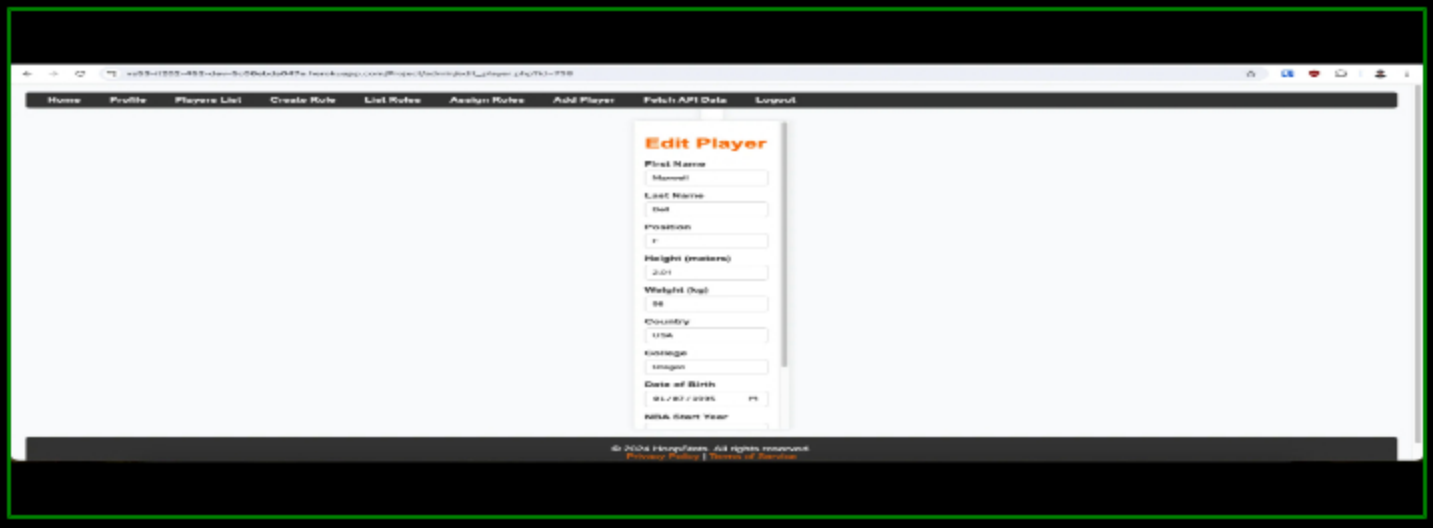
Caption (required) ✓

Describe/highlight what's being shown

Screenshot of green "Player updated successfully message"

#4) Design/Style should be considered (i.e. bootstrap, custom css, etc)

#4) Design/Style should be considered (i.e., Bootstrap, custom css, etc)



Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing the design of the page

Explanation (required) ✓

Briefly explain your design choices

PREVIEW RESPONSE

For the design of the edit page, I wanted to keep it simple and user-friendly, matching the overall theme of the website. I used a clean layout with a black and orange color scheme to reflect the basketball theme. The form fields are clearly labeled and aligned, making it easy to read and fill out. I used CSS to style the input fields and buttons, giving them a consistent look and feel. The navigation bar is easily accessible, and I used responsive design principles to make the page look good.

COLLAPSE

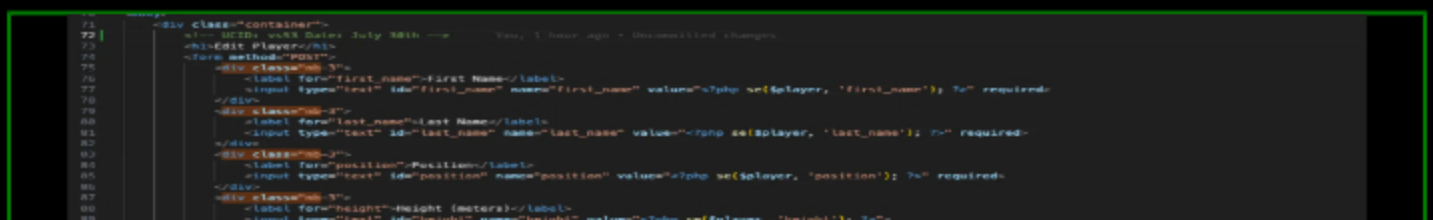
Task #2 - Points: 1

Text: Screenshots of edit page code

Details:

Include ucid/date comments for each code screenshot

#1) Form should have correct data types for each property being requested



Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing the form will have correct data types for each property

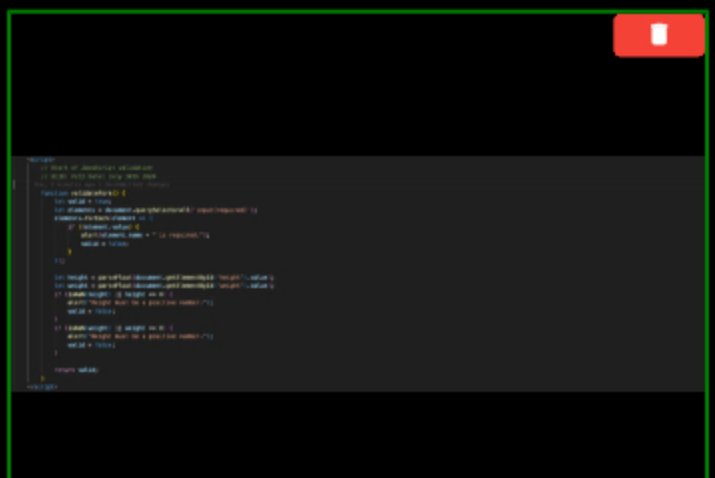
Explanation (required) ✓

Briefly explain the data type choices

PREVIEW RESPONSE

In the form, I chose data types that match the kind of information we're collecting for each player. For example, I used "text" for names and positions because they are typically words. For height and weight, I selected "number" to make sure the input is numeric and added constraints like step values for precision. The "date" type was used for the birth date to help users pick a valid date easily. For years pro and NBA start year, I also went with "number" to keep these fields numeric. This setup helps keep the data accurate and consistent.

#2) Form should have correct validation for each field (HTML, JS, and PHP)



Caption (required) ✓

Describe/highlight what's being shown

3 screenshots showing the HTML, JS, and PHP validations

Explanation (required) ✓

Briefly explain the validations

PREVIEW RESPONSE

I implemented validations to make sure the data entered is correct. The HTML validations require fields to be filled out and use types like "number" and "date" to restrict input. The JavaScript validation checks fields before the form is submitted, providing immediate feedback if something is wrong, like non-numeric input for numbers. On the server side, PHP validations catch any errors missed by the client-side checks, such as making sure height and weight are positive numbers and that required fields are not empty. This approach catches errors early and prevents bad data from being saved.

#3) Successful edit should have a user-friendly message

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing that the edit will show a user-friendly message

Explanation (required) ✓

Provide a high-level step-by-step of how the fetching of the record, populating the form, and the update works and gets changed in your DB

PREVIEW RESPONSE

When a player's information is successfully edited, the process is straightforward. First, the form data is submitted, and the server fetches the existing record using the player's ID. The form is then populated with this data, allowing the user to make changes. Upon submission, the updated data is validated and then saved back to the database. If everything goes well, a user-friendly message like "Player updated successfully" is displayed to confirm the edit. This message provides clear feedback that the changes were saved.

#4) Any errors should have user-friendly messages

Caption (required) ✓

Describe/highlight what's being shown

Screenshot showing that any errors will have a user-friendly message for each scenario

Explanation (required) ✓

Briefly explain the possible errors

PREVIEW RESPONSE

Errors can happen if the user submits incorrect or incomplete data. For instance, leaving required fields empty or entering non-numeric values in number fields triggers error messages. These messages are displayed as friendly alerts, like "First name is required" or "Height must be a positive number." This way, users know exactly what went wrong and can correct it.

#5) Include any other rules like role guards and login checks

```
// UCID: Vx53 Date: July 30th 2024

// Ensure the user is logged in and has admin privileges
is_logged_in(true);
if (!has_role("Admin")) {
    flash("You do not have permission to access this page", "danger");
    die(header("Location: " . get_url("home.php")));
}
```

Caption (required) ✓

Describe/highlight what's being shown
Screenshot showing role guards and checks for admin

Explanation (required) ✓
Briefly explain the logic/reasoning
PREVIEW RESPONSE
The edit player page has role guards and login checks to protect it. Only logged-in users with admin privileges can access this page. This is crucial because not everyone should have the ability to edit player information. If someone without the right permissions tries to access the page, they are redirected with an appropriate message.

COLLAPSE

Task #3 - Points: 1

Text: Screenshot of records from DB

#1) Show a before and after screenshot of the record (two screenshots)

Caption (required) ✓
Describe/highlight what's being shown
Showing the player Jordan Bell, as the example I used previously, and the difference in the database records

Explanation (required) ✓
Explain what differs
PREVIEW RESPONSE
The only thing I changed was the name. Previously it was "Jordan Bell" and now I changed it to be "Maxwell Bell" So you can see in the first name column, the name is different.

COLLAPSE

Task #4 - Points: 1

Text: Add related links

Checklist

*The checkboxes are for your own tracking

#	Points	Details
#1	1	Include the heroku prod link for this page
#2	1	Add the pull request link for the branch related to this feature Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature

URL #1
https://vs53-it202-452-prod-1c77a1c304b2.herokuapp.com/Project/admin/edit_player.php?id=738

URL
https://vs53-it202-452-prod-1c77a1c304b2.herol

URL #2
https://github.com/VikShah/vs53-it202-452/pull/48

URL
https://github.com/VikShah/vs53-it202-452/pull

+ ADD ANOTHER URL

Delete Handling (1 pt.)

^COLLAPSE^

^COLLAPSE^

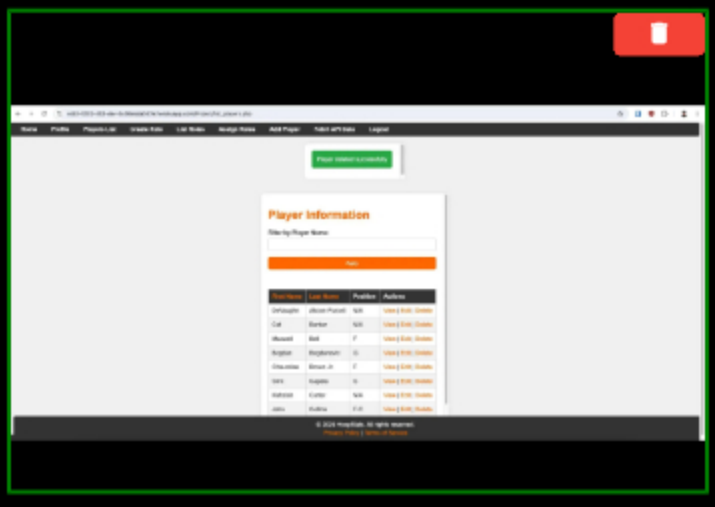
Task #1 - Points: 1

Text: Screenshots related to delete

Details:
Heroku dev url must be visible in all relevant screenshots

Include ucid/date comments for each code screenshot

#1) Show the success message of a delete

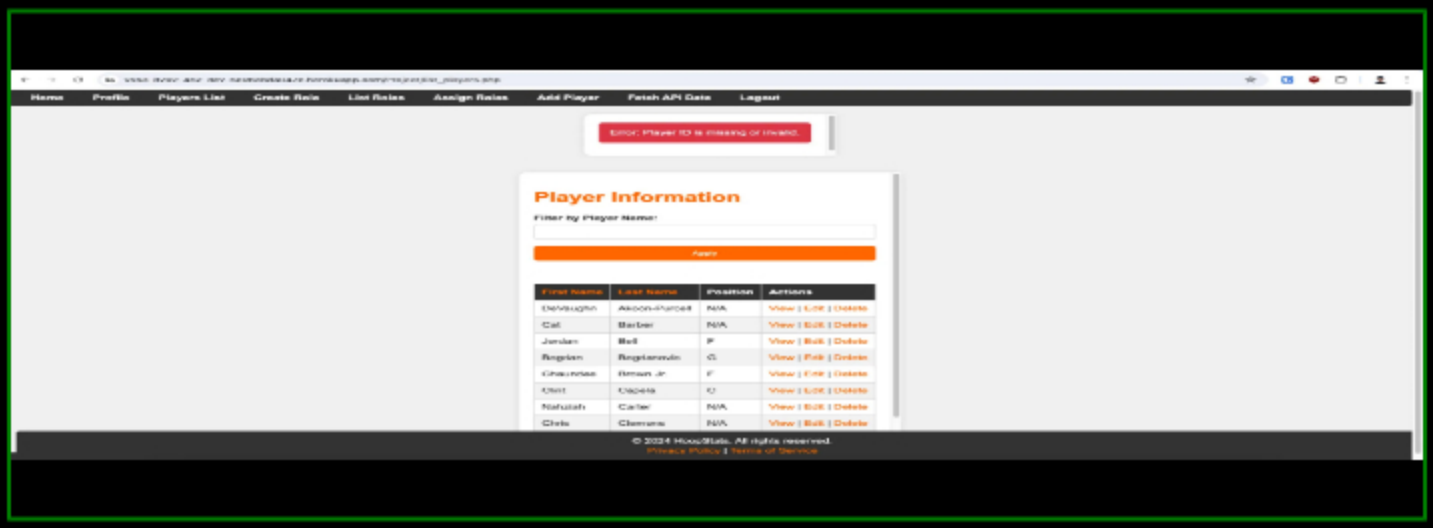


Caption (required) ✓

Describe/highlight what's being shown

Two screenshots showing the process of deleting a player

#2) Show any error messages of a failed delete (like id not being passed)



Caption (required) ✓

Describe/highlight what's being shown

The screenshot shows an error message displayed when a delete request is made without a valid player ID

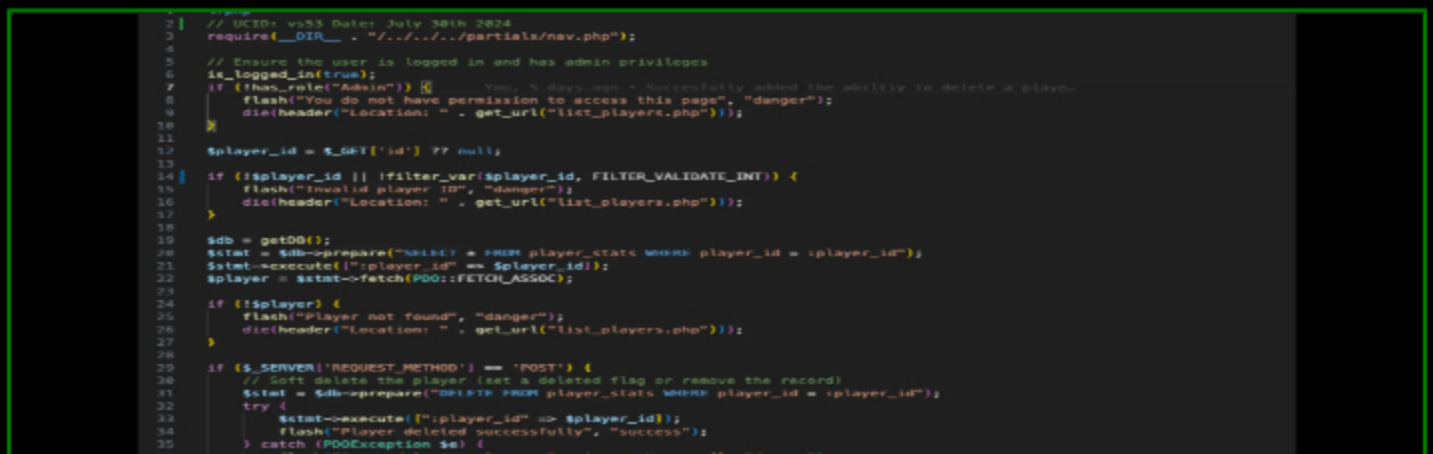
Explanation (required) ✓

Explain in concise steps how this logically works

PREVIEW RESPONSE

In the delete_player.php file, I included a check to see if a valid player ID is provided in the URL. If the player ID is missing or invalid, the code triggers a flash message that says "Invalid player ID" and then redirects back to the players list page. This way, users are informed about the issue and are taken back to the list of players without any changes being made. The logic checks the ID, and if it's not correct, it stops further processing and shows the error message.

#3) Show the code related to the delete processing



```php
// UCID: vs55 Date: July 30th 2024
require(__DIR__ . "/../../../partials/nav.php");

// Ensure the user is logged in and has admin privileges
is_logged_in(true);
if (!has_role("Admin")) {
    flash("You do not have permission to access this page", "danger");
    die(header("Location: " . get_url("list_players.php")));
}

$player_id = $_GET['id'] ?? null;

if (!$player_id || !filter_var($player_id, FILTER_VALIDATE_INT)) {
    flash("Invalid player ID", "danger");
    die(header("Location: " . get_url("list_players.php")));
}

$db = getDB();
$stmt = $db->prepare("SELECT * FROM player_stats WHERE player_id = :player_id");
$stmt->execute([":player_id" => $player_id]);
$player = $stmt->fetch(PDO::FETCH_ASSOC);

if (!$player) {
    flash("Player not found", "danger");
    die(header("Location: " . get_url("list_players.php")));
}

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // Soft delete the player (set a deleted flag or remove the record)
    $stmt = $db->prepare("DELETE FROM player_stats WHERE player_id = :player_id");
    try {
        $stmt->execute([":player_id" => $player_id]);
        flash("Player deleted successfully", "success");
    } catch (PDOException $e) {
```

Caption (required) ✓

Describe/highlight what's being shown

The screenshot shows the code that handles the deletion of a player record. It highlights the PHP code responsible for it

Explanation (required) ✓

Explain in concise steps how this logically works

PREVIEW RESPONSE

In the delete_player.php file, the delete processing starts with a check to confirm the user is logged in and has admin privileges. It retrieves the player ID from the URL and checks if it is valid. If the player ID is valid, the code retrieves the player record from the database. Upon confirming the player's existence, the script waits for a POST request to delete the player. When the delete button is clicked, it executes a SQL DELETE query to remove the player from the player_stats table. After successful deletion, a flash message "Player deleted successfully" is shown, and the user is redirected back to the players list page. This process ensures only valid and authorized deletions.

#4) Explain the delete logic

Explanation (required) ✓

Is it a soft or hard delete? Are there any necessary roles or restrictions? (can only delete their data, can only be done by admin, etc)?

PREVIEW RESPONSE

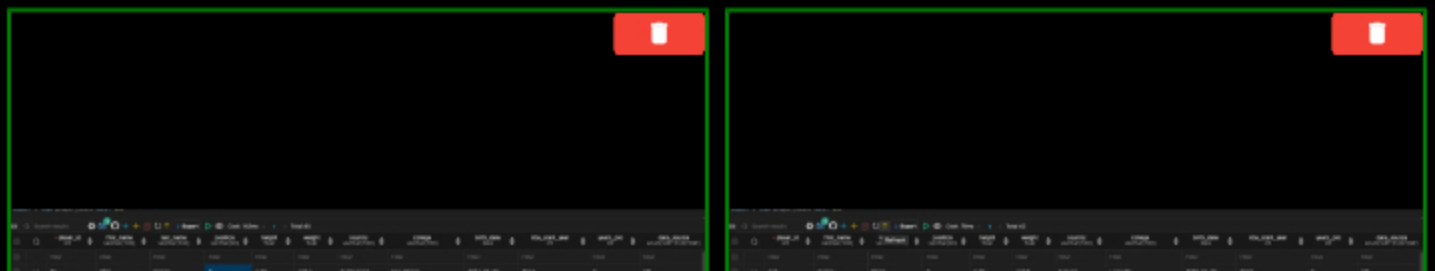
The delete logic in my project is a hard delete, meaning that the player's data is permanently removed from the database. Only users with admin privileges can perform deletions. This restriction is necessary to prevent unauthorized users from deleting important data. Before the deletion occurs, the code checks if the user is logged in and has the admin role. If these conditions are met, the code proceeds to delete the player record from the player_stats table. This setup ensures that only authorized personnel can make such significant changes.

COLLAPSE

Task #2 - Points: 1

Text: Screenshots of the data

#1) Show a before and after screenshot of the DB data (two screenshots)



Caption (required) ✓

Describe/highlight what's being shown (note precisely what changed)

Showing before and after for "Clint capela". Showing the screenshot of the 2 databases before and after

COLLAPSE

Task #3 - Points: 1

Text: Add the pull request link for the branch related to this feature

Details:

Note: the link should end with /pull/#. Same pull request shouldn't be used for each feature

URL #1

https://github.com/VikShah/vs53-it202-452/pull/49

URL

https://github.com/VikShah/vs53-it202-452/pull,

+ ADD ANOTHER URL

COLLAPSE

Misc (1 pt.)

COLLAPSE

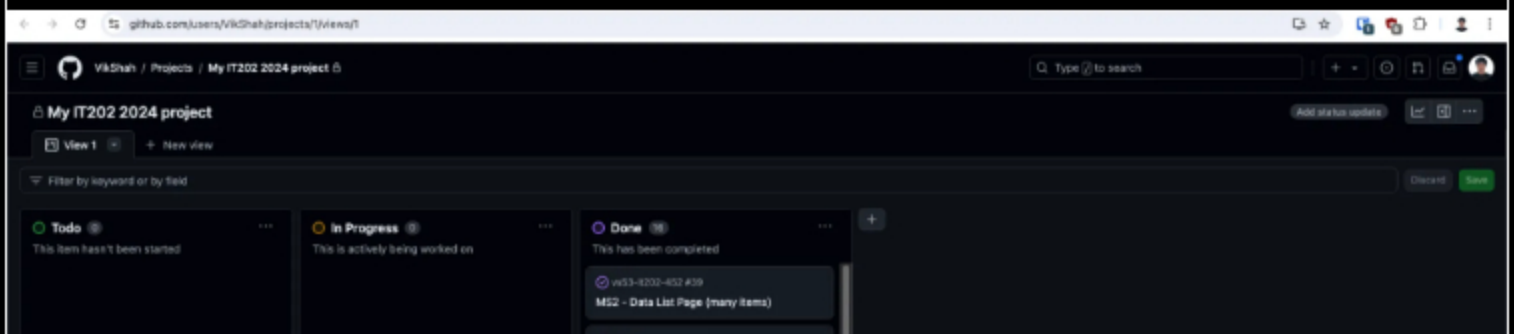
Task #1 - Points: 1

Text: Screenshot of your project board from GitHub (tasks should be in the proper column)

Task Screenshots:

Gallery Style: Large View

Small Medium Large



Screenshot showing my github project, and all the MS2 objectives

COLLAPSE

Task #2 - Points: 1

Text: Provide a direct link to the project board on GitHub

URL #1
https://github.com/users/VikShah/projects/1/views/1

URL
https://github.com/users/VikShah/projects/1/vi

+ ADD ANOTHER URL

COLLAPSE

Task #3 - Points: 1

Text: Talk about any issues or learnings during this assignment

Response:

During this assignment, I faced a few challenges and learned a lot. One major issue was handling errors and making sure user-friendly messages were displayed. I had to understand how to properly use PHP sessions and flash messages. Working with API data and integrating it into my database was another learning curve, especially with handling duplicate data and ensuring the data was accurate. I also had to improve my understanding of role-based access control to restrict certain actions to admin users only. Overall, this assignment taught me the importance of thorough validation and error handling, as well as the complexities of user permissions and data management in web applications.

COLLAPSE

Task #4 - Points: 1

Text: WakaTime Screenshot

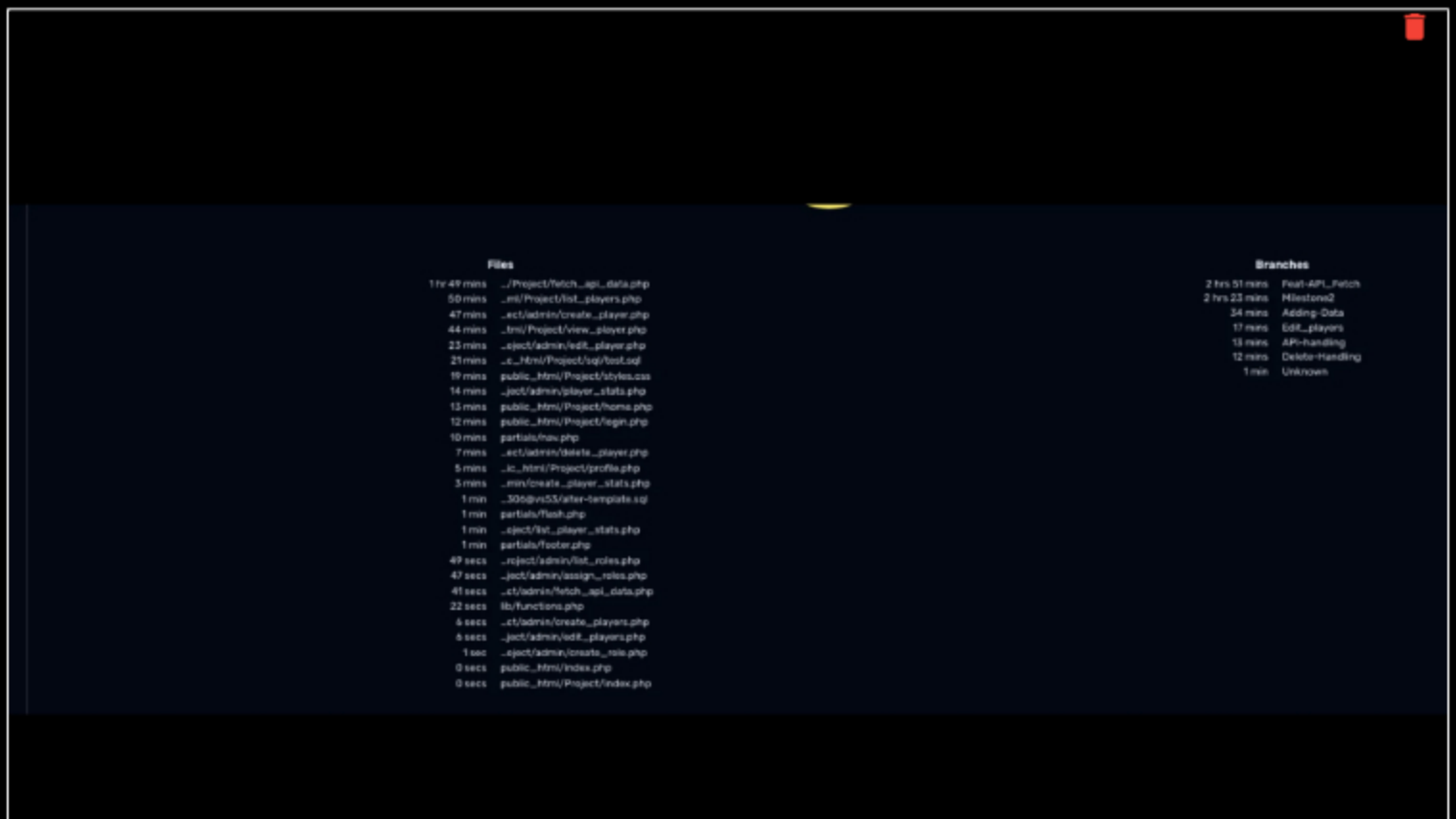
Details:
Grab a snippet showing the approximate time involved that clearly shows your repository. The duration isn't considered for grading, but there should be some time involved

Task Screenshots:

Gallery Style: Large View

Small Medium Large

Showing how many overs over the last week



Showing more detail of how much I spent on each branches

End of Assignment