ASSETS > CSS > 1

**1. Main Body (CSS for <body>):**

* **display: flex;**: The body uses a flexbox layout, allowing elements (like the sidebar and content) to align side-by-side.
* **margin: 0;**: Removes the default margin around the page.
* **font-family: Arial, sans-serif;**: Sets a clean and readable font for the text.

**2. Sidebar (.sidebar):**

* **position: fixed; top: 0; left: 0;**: The sidebar is "stuck" to the top-left corner of the viewport and won't move even when scrolling.
* **width: 250px;**: The sidebar is always 250 pixels wide.
* **height: 100vh;**: It spans the full height of the browser window.
* **background-color: #9c1515; color: white;**: Gives the sidebar a dark red background with white text.
* **display: flex; flex-direction: column;**: Its elements are stacked vertically (column direction).
* **justify-content: space-between;**: The items inside are spaced out, with the header at the top, content in the middle, and footer at the bottom.

**3. Sidebar Sections:**

* **Header (.sidebar-header):**
  + Displays at the top of the sidebar.
  + Contains an image centered both vertically and horizontally.
  + The image automatically resizes to fit within half the width of the sidebar.
* **Content (.sidebar-content):**
  + Holds the main items (e.g., navigation links or menu options).
  + Starts slightly higher than usual because of the negative margin-top: -151px;.
* **Footer (.sidebar-footer):**
  + Displays at the bottom of the sidebar.
  + Useful for additional links or copyright info.

**4. Menu Items (.menu-item):**

* **padding: 15px;**: Adds space inside the menu items for a larger clickable area.
* **cursor: pointer;**: Changes the cursor to a pointer (hand) on hover.
* **text-decoration: none;**: Removes underlines from text links.
* **Hover effect:** When a user hovers over a menu item, it changes its background color to a slightly transparent pinkish-red (#ffdcdca4).

**5. Main Content Area (.content):**

* **flex: 1;**: This area takes up the remaining available width after accounting for the sidebar.
* **padding: 20px;**: Adds some space around the content.
* **background-color: #f4f4f4;**: Gives it a light gray background.
* **margin-left: 250px;**: Creates space on the left to align with the sidebar.

**What It Looks Like:**

* A **fixed sidebar** on the left with a header, menu items, and a footer.
* A **main content area** on the right that adjusts its size based on the window width.
* The sidebar stays visible while the content scrolls.

This structure is common for dashboards or apps with navigation menus on the side.

ASSETS >JS

This JavaScript code dynamically generates a **calendar** and allows users to navigate through months. Here's a breakdown of how it works:

**1. Event Listener for DOM Content Loaded**

document.addEventListener('DOMContentLoaded', () => { ... });

* This ensures that the script runs only after the HTML document is fully loaded and ready.
* All calendar functionality is encapsulated within this listener.

**2. Generate Calendar Function**

function generateCalendar(month, year) { ... }

* **Purpose:** Dynamically creates the calendar for a given month and year.
* **Steps Inside the Function:**
  1. **Get the Calendar Body:**
     + const calendarBody = document.getElementById('calendarBody');
     + Clears the current content using calendarBody.innerHTML = '';.
  2. **Calculate Days in Month:**
     + new Date(year, month + 1, 0).getDate() calculates the number of days in the specified month.
  3. **Create Day Cells:**
     + A for loop iterates through the days of the month.
     + For each day, a div is created, styled with day-cell and bg-light classes, and displays the day number.
  4. **Special Indicator for June 9:**
     + If the date is June 9 (month === 5 && i === 9), a special indicator (booking-indicator) is added to the day.
     + This cell gets a booking class for styling.
  5. **Append Cells:**
     + Adds the created day cell to the calendarBody.

**3. Update the Current Month and Year Display**

const currentMonthYear = document.getElementById('currentMonthYear');

const monthNames = ["January", "February", ... , "December"];

currentMonthYear.textContent = `${monthNames[month]} ${year}`;

* The currentMonthYear element is updated to display the current month name and year.
* Uses an array of month names for easy reference.

**4. Handle Next/Previous Month Navigation**

**Next Month Button:**

document.getElementById('nextBtn').addEventListener('click', () => { ... });

* **Logic:**
  + Increments the currentMonth.
  + If it exceeds 11 (December), resets to 0 (January) and increments the year.
  + Calls generateCalendar() with updated values.

**Previous Month Button:**

document.getElementById('prevBtn').addEventListener('click', () => { ... });

* **Logic:**
  + Decrements the currentMonth.
  + If it goes below 0 (January), resets to 11 (December) and decrements the year.
  + Calls generateCalendar() with updated values.

**Output: What It Does**

1. **Dynamic Calendar:** Displays a calendar for the current month and year by default.
2. **Day Highlighting:** Special styling and indicator for a specific date (June 9).
3. **Navigation:**
   * Buttons (nextBtn and prevBtn) allow users to navigate between months.
   * Automatically handles month/year boundaries (e.g., December to January).

**Key Elements in HTML**

Ensure the following HTML structure exists for the script to work:

<div id="currentMonthYear"></div>

<div id="calendarBody"></div>

<button id="prevBtn">Previous</button>

<button id="nextBtn">Next</button>

**Usage Scenarios**

* Event scheduling tools.
* Date-picking UI for forms.
* Visual representation of bookings or availability.

If you'd like help styling the calendar or extending its functionality (e.g., event handling for specific dates), let me know!

.GIT > CONFIG

This is a **Git configuration file (.git/config)** for a repository. Here's what each section means:

**1. Core Settings ([core]):**

* **repositoryformatversion = 0**: Indicates the repository format version. This is typically 0 for most repositories.
* **filemode = false**: Disables file permission tracking. This is useful on systems like Windows where file permissions are handled differently.
* **bare = false**: Indicates this is a working repository (not bare). A bare repository is used for sharing (e.g., as a remote repository) and has no working directory.
* **logallrefupdates = true**: Ensures updates to references (branches, tags) are logged for history.
* **symlinks = false**: Disables symbolic links in the repository. Useful on systems that don't support symlinks well (e.g., Windows).
* **ignorecase = true**: Makes Git ignore case differences in file names. Again, helpful on case-insensitive file systems like Windows.

**2. Submodule Settings ([submodule]):**

* **active = .**: Indicates a submodule is active. Submodules allow including one Git repository as a subdirectory of another.

**3. Remote Repository Settings ([remote "origin"]):**

* **url = https://github.com/100011000/Lucky3tours.git**: Specifies the remote repository's URL. This is where the repository is pushed/pulled from.
* **fetch = +refs/heads/\*:refs/remotes/origin/\***: Maps all branches in the remote repository (refs/heads/\*) to local tracking branches (refs/remotes/origin/\*).

**4. Branch Settings ([branch "main"]):**

* **remote = origin**: Associates the main branch with the origin remote.
* **merge = refs/heads/main**: Specifies the branch in the remote repository to merge with the local main branch during pulls.

**5. Git Large File Storage (LFS) Settings ([lfs]):**

* **repositoryformatversion = 0**: Indicates the repository supports Git LFS (Large File Storage). Git LFS manages large files by storing pointers instead of the files themselves in the repository.

**Purpose of This File**

This file:

1. Configures how Git behaves for this repository.
2. Defines the connection to a remote repository hosted on GitHub.
3. Specifies settings for handling branches, submodules, and large files.

If you'd like help modifying this configuration or understanding specific parts, let me know!

FUNCTIONS > ADD CUSTOMER

This PHP script is used to process a form submission for creating a new bus reservation entry in a database. Here's a breakdown of what each part of the script does:

**1. Include Database Connection**

require\_once('../pages/db\_connect.php');

* This line includes a file that establishes the database connection ($conn).
* Ensure the file path (../pages/db\_connect.php) is correct relative to the location of this script.

**2. Handle POST Request**

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') { ... }

* This checks if the request method is POST, ensuring the script only runs when the form is submitted.

**3. Retrieve Form Data**

$name = $\_POST['Name'];

$email = $\_POST['Email'];

$contact\_no = $\_POST['Contact\_No'];

$passenger\_qty = $\_POST['Passenger\_qty'];

$reserved\_date = $\_POST['Reserved\_date'];

$bus\_type = $\_POST['Bus\_Type'];

$destination = $\_POST['Destination'];

$drivers = $\_POST['Drivers'];

$status = $\_POST['Status'];

* Retrieves data from the submitted form using $\_POST.
* The keys (e.g., Name, Email) must match the name attributes in the form fields.

**4. Determine Bus Price**

if ($bus\_type === 'Bus') {

$bus\_price = 20000;

} else if ($bus\_type === 'Coaster') {

$bus\_price = 10000;

} else {

$bus\_price = 0;

}

* Sets the price based on the selected bus type.
* Defaults to 0 if the bus type is unrecognized.

**5. Insert Data into the Database**

$sql = "INSERT INTO bus\_customer\_tbl (Name, Email, Contact\_No, Passenger\_qty, Reserved\_date, Bus\_Type, Bus\_Price, Destination, Drivers, Status)

VALUES ('$name', '$email', '$contact\_no', '$passenger\_qty', '$reserved\_date', '$bus\_type', '$bus\_price','$destination','$drivers','$status')";

* Builds an SQL query to insert the form data into the bus\_customer\_tbl table.
* Columns in the table (Name, Email, etc.) should match the database schema.

**6. Execute Query**

if ($conn->query($sql) === TRUE) {

header('Location: ../pages/customers.php');

exit();

} else {

echo 'Error: ' . $conn->error;

exit();

}

* Executes the query using $conn->query($sql).
* If successful:
  + Redirects to a success page (../pages/customers.php).
* If there's an error:
  + Outputs the error using $conn->error.

**7. Close the Connection**

$conn->close();

* Ensures the database connection is closed after the query is executed.

**Improvements & Security Suggestions**

1. **SQL Injection Protection:**
   * Use **prepared statements** to avoid SQL injection.
2. $stmt = $conn->prepare("INSERT INTO bus\_customer\_tbl (Name, Email, Contact\_No, Passenger\_qty, Reserved\_date, Bus\_Type, Bus\_Price, Destination, Drivers, Status) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
3. $stmt->bind\_param("ssssssssss", $name, $email, $contact\_no, $passenger\_qty, $reserved\_date, $bus\_type, $bus\_price, $destination, $drivers, $status);
4. $stmt->execute();
5. **Input Validation:**
   * Validate and sanitize input data to ensure it's in the correct format (e.g., email validation, numeric checks for Passenger\_qty).
6. **Error Handling:**
   * Provide user-friendly error messages rather than raw database errors.
7. **Environment Variables for Database Credentials:**
   * Store sensitive data (e.g., database credentials) in environment variables instead of hardcoding them in db\_connect.php.
8. **Use HTTPS:**
   * Ensure the form submission is over HTTPS to encrypt data in transit.

If you'd like help implementing any of these improvements, let me know!

FUNCTIONS > BOOKING FETCH

This PHP script performs two main tasks: fetching data from the database and inserting a new entry into the bus\_customer\_tbl table. Here’s a detailed explanation:

**1. Include Database Connection**

require\_once('../pages/db\_connect.php');

* Includes the file that establishes a database connection ($conn).
* Ensure the path (../pages/db\_connect.php) is correct relative to this script.

**2. Check Request Method**

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') { ... }

* Ensures the script runs only when the request method is POST (usually from a form submission).

**3. Retrieve Input Data**

$name = $\_POST['Name'];

$customers\_id = $\_POST['Customers\_Id'];

$check\_in = $\_POST['Reserved\_date'];

$check\_out = $\_POST['Reserved\_date'];

$price = $\_POST['Bus\_Price'];

* Retrieves form data using $\_POST.
* The keys (e.g., Name, Customers\_Id) should match the name attributes in the form.

**4. Fetch Customer Details**

$sql1 = "SELECT \* FROM bus\_customer\_tbl WHERE Customers\_Id = '".$customers\_id."'";

$result = $conn->query($sql1);

$row = $result->fetch\_assoc();

$bus\_type = $row['Bus\_Type'];

$bus\_price = $row['Bus\_Price'];

* Fetches data for a specific customer from the bus\_customer\_tbl table using the Customers\_Id.
* fetch\_assoc() retrieves the first matching row as an associative array.

**5. Insert a New Record**

$sql = "INSERT INTO `bus\_customer\_tbl` (name, Bus\_Type, Customers\_Id, Reserved\_date, price) VALUES ('$name', '$bus\_type', '$customers\_id', '$check\_in', '$price')";

* Inserts the new record into the bus\_customer\_tbl table.
* The columns in the query (name, Bus\_Type, etc.) must match those in your database schema.

**6. Execute the Query**

if ($conn->query($sql) === TRUE) {

header('Location: ../pages/customers.php');

exit();

} else {

echo 'Error: ' . $conn->error;

exit();

}

* If the query executes successfully, redirects the user to a customers page (../pages/customers.php).
* If there’s an error, it displays the error message.

**7. Close the Connection**

$conn->close();

* Closes the database connection after executing the script.

**Issues and Improvements**

**1. Duplicate Insertions**

* You're inserting into the same table (bus\_customer\_tbl) where you're fetching data. This could lead to duplicate entries or unexpected behavior. Consider separating the table used for storing bookings.

**2. Security: SQL Injection Risk**

* Using raw string concatenation in SQL queries opens the application to SQL injection attacks. Replace this with prepared statements:

$stmt = $conn->prepare("INSERT INTO bus\_customer\_tbl (name, Bus\_Type, Customers\_Id, Reserved\_date, price) VALUES (?, ?, ?, ?, ?)");

$stmt->bind\_param("ssiss", $name, $bus\_type, $customers\_id, $check\_in, $price);

$stmt->execute();

**3. Data Validation and Sanitization**

* Validate inputs to ensure they meet expected formats (e.g., numeric for Customers\_Id and Bus\_Price, valid date format for Reserved\_date).
* Sanitize user inputs using filter\_var() or similar functions to avoid malicious data.

**4. Logical Error: Reserved\_date for check\_in and check\_out**

* You’re using the same Reserved\_date value for both check\_in and check\_out. Clarify whether this is intentional or requires distinct inputs.

**5. Potential Logic Issue**

* Fetching Bus\_Type and Bus\_Price from the bus\_customer\_tbl table assumes they’re already present. Ensure the Customers\_Id exists in the database; otherwise, $row might be null, leading to errors.

**Refactored Code (Prepared Statements)**

<?php

require\_once('../pages/db\_connect.php');

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

$name = $\_POST['Name'];

$customers\_id = $\_POST['Customers\_Id'];

// Fetch customer data

$stmt1 = $conn->prepare("SELECT Bus\_Type, Bus\_Price FROM bus\_customer\_tbl WHERE Customers\_Id = ?");

$stmt1->bind\_param("s", $customers\_id);

$stmt1->execute();

$result = $stmt1->get\_result();

if ($row = $result->fetch\_assoc()) {

$bus\_type = $row['Bus\_Type'];

$bus\_price = $row['Bus\_Price'];

// Insert new reservation

$check\_in = $\_POST['Reserved\_date'];

$price = $\_POST['Bus\_Price'];

$stmt2 = $conn->prepare("INSERT INTO bus\_customer\_tbl (name, Bus\_Type, Customers\_Id, Reserved\_date, price) VALUES (?, ?, ?, ?, ?)");

$stmt2->bind\_param("ssiss", $name, $bus\_type, $customers\_id, $check\_in, $price);

if ($stmt2->execute()) {

header('Location: ../pages/customers.php');

exit();

} else {

echo "Error: " . $stmt2->error;

exit();

}

} else {

echo "Error: Customer ID not found.";

exit();

}

$conn->close();

}

?>

This version addresses SQL injection risks and provides better error handling. Let me know if you need further assistance!

FUNCTION > DELETE CUSTOMERS

This PHP script allows for the deletion of a record from the bus\_customer\_tbl table based on the Customers\_Id value passed in a POST request. Here's a breakdown:

**1. Include the Database Connection**

require\_once('../pages/db\_connect.php');

* Includes the database connection file (db\_connect.php) to access the database using $conn.

**2. Validate the Request Method**

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

* Ensures the script processes only POST requests, typically coming from a form submission.

**3. Validate Input**

if (isset($\_POST['Customers\_Id'])) {

$customers\_id = $\_POST['Customers\_Id'];

* Checks if Customers\_Id is provided in the request.
* Assigns the value to the $customers\_id variable for further use.

**4. Prepare the DELETE Query**

$sql = "DELETE FROM bus\_customer\_tbl WHERE Customers\_Id = ?";

$stmt = $conn->prepare($sql);

* Uses a **prepared statement** to prevent SQL injection.
* The ? is a placeholder for the Customers\_Id value.

**5. Bind and Execute the Statement**

$stmt->bind\_param("i", $customers\_id);

if ($stmt->execute()) {

header('Location: ../pages/customers.php');

exit();

} else {

echo 'Error executing query: ' . $stmt->error;

}

* **bind\_param("i", $customers\_id)**: Binds the Customers\_Id value to the query as an integer (i).
* Executes the query and checks for success:
  + On success: Redirects the user to ../pages/customers.php.
  + On failure: Displays an error message.

**6. Handle Errors**

* **Preparation Failure:**

echo 'Error preparing query: ' . $conn->error;

Displays an error message if the query preparation fails.

* **Missing Customers\_Id:**

echo 'Error: booking ID is not set.';

Displays an error if Customers\_Id is not provided in the request.

* **Invalid Request Method:**

echo 'Error: invalid request method.';

Ensures only POST requests are processed.

**7. Close Resources**

$stmt->close();

$conn->close();

* Closes the prepared statement and database connection to free up resources.

**Potential Improvements**

1. **Input Validation:**
   * Validate and sanitize Customers\_Id to ensure it is numeric:
   * if (filter\_var($customers\_id, FILTER\_VALIDATE\_INT) === false) {
   * echo 'Invalid Customers\_Id.';
   * exit();
   * }
2. **User Feedback:**
   * Display a success or failure message on the redirection page.
3. **Error Logging:**
   * Instead of showing error messages directly, log errors to a file for security and debugging:
   * error\_log('Error executing query: ' . $stmt->error, 3, 'errors.log');
4. **Redirect on Failure:**
   * Redirect to an error page instead of displaying raw errors:
   * header('Location: ../pages/error.php?error=DeletionFailed');
   * exit();
5. **CSRF Protection:**
   * Include a CSRF token to protect against unauthorized requests.

**Refactored Code**

<?php

require\_once('../pages/db\_connect.php'); // Ensure this path is correct

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

if (isset($\_POST['Customers\_Id'])) {

$customers\_id = $\_POST['Customers\_Id'];

// Validate Customers\_Id

if (filter\_var($customers\_id, FILTER\_VALIDATE\_INT) === false) {

echo 'Invalid Customers\_Id.';

exit();

}

// Prepare the DELETE statement

$sql = "DELETE FROM bus\_customer\_tbl WHERE Customers\_Id = ?";

$stmt = $conn->prepare($sql);

if ($stmt) {

// Bind the Customers\_Id to the statement

$stmt->bind\_param("i", $customers\_id);

// Execute the statement

if ($stmt->execute()) {

// Redirect to the customers page after successful deletion

header('Location: ../pages/customers.php');

exit();

} else {

error\_log('Error executing query: ' . $stmt->error, 3, '../logs/errors.log');

header('Location: ../pages/error.php?error=DeletionFailed');

exit();

}

$stmt->close();

} else {

error\_log('Error preparing query: ' . $conn->error, 3, '../logs/errors.log');

header('Location: ../pages/error.php?error=PreparationFailed');

exit();

}

$conn->close();

} else {

header('Location: ../pages/error.php?error=MissingCustomersId');

exit();

}

} else {

header('Location: ../pages/error.php?error=InvalidRequest');

exit();

}

?>

This version includes better validation, error handling, and user feedback while maintaining security. Let me know if you need further assistance!

FUNCTIONS > EDIT CUSTOMER

This PHP script handles the updating of customer records in the bus\_customer\_tbl table. Here's a breakdown of its functionality:

**Overview**

The script:

1. Updates a customer record in the database based on the provided Customers\_Id.
2. Redirects to a success page if the update is successful or displays an error message otherwise.

**Code Breakdown**

**1. Include Database Connection**

require\_once('../pages/db\_connect.php');

* Ensures the database connection is established using $conn.

**2. Check for POST Request**

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

* Processes the form submission only if the request method is POST.

**3. Collect Data from the Form**

$customer\_id = $\_POST['Customers\_Id'];

$name = $\_POST['Name'];

$email = $\_POST['Email'];

$contact\_no = $\_POST['Contact\_No'];

$passenger\_qty = $\_POST['Passenger\_qty'];

$reserved\_date = $\_POST['Reserved\_date'];

$bus\_type = $\_POST['Bus\_Type'];

$destination = $\_POST['Destination'];

$drivers = $\_POST['Drivers'];

$status = $\_POST['Status'];

* Collects data submitted via the form using $\_POST.

**4. Construct the Update Query**

$sql = "UPDATE bus\_customer\_tbl SET

Name = '$name',

Email = '$email',

Contact\_No = '$contact\_no',

Passenger\_qty = '$passenger\_qty',

Reserved\_date = '$reserved\_date',

Bus\_Type = '$bus\_type',

Destination = '$destination',

Drivers = '$drivers',

Status = '$status'

WHERE Customers\_Id = '$customer\_id'";

* Updates the record in the bus\_customer\_tbl table where Customers\_Id matches the provided ID.

**5. Execute the Query**

if ($conn->query($sql) === TRUE) {

header('Location: ../pages/customers.php');

exit();

} else {

echo 'Error updating record: ' . $conn->error;

exit();

}

* If the query is successful:
  + Redirects to ../pages/customers.php (update the path as needed).
* If the query fails:
  + Displays an error message with details from $conn->error.

**6. Close the Connection**

$conn->close();

* Ensures the database connection is properly closed.

**Potential Issues and Fixes**

**1. SQL Injection**

* Directly inserting user inputs into the SQL query makes it vulnerable to SQL injection attacks.
* **Fix: Use Prepared Statements.**

$sql = "UPDATE bus\_customer\_tbl SET

Name = ?,

Email = ?,

Contact\_No = ?,

Passenger\_qty = ?,

Reserved\_date = ?,

Bus\_Type = ?,

Destination = ?,

Drivers = ?,

Status = ?

WHERE Customers\_Id = ?";

$stmt = $conn->prepare($sql);

$stmt->bind\_param(

"sssssssssi",

$name,

$email,

$contact\_no,

$passenger\_qty,

$reserved\_date,

$bus\_type,

$destination,

$drivers,

$status,

$customer\_id

);

if ($stmt->execute()) {

header('Location: ../pages/customers.php');

exit();

} else {

echo 'Error updating record: ' . $stmt->error;

exit();

}

$stmt->close();

$conn->close();

**2. Input Validation**

* **Validate user input** to ensure it meets the expected format (e.g., email validation, numeric values for quantities).

if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

echo 'Invalid email format.';

exit();

}

if (!is\_numeric($contact\_no)) {

echo 'Contact number must be numeric.';

exit();

}

**3. Error Handling**

* **Current Issue**: Error messages are shown directly to the user.
* **Fix**: Redirect to an error page or log errors.

error\_log('Error updating record: ' . $conn->error, 3, '../logs/errors.log');

header('Location: ../pages/error.php?error=UpdateFailed');

exit();

**Refactored Code**

Here's the improved version:

<?php

require\_once('../pages/db\_connect.php');

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

$customer\_id = $\_POST['Customers\_Id'];

$name = $\_POST['Name'];

$email = $\_POST['Email'];

$contact\_no = $\_POST['Contact\_No'];

$passenger\_qty = $\_POST['Passenger\_qty'];

$reserved\_date = $\_POST['Reserved\_date'];

$bus\_type = $\_POST['Bus\_Type'];

$destination = $\_POST['Destination'];

$drivers = $\_POST['Drivers'];

$status = $\_POST['Status'];

// Validate email

if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

header('Location: ../pages/error.php?error=InvalidEmail');

exit();

}

// Use prepared statements

$sql = "UPDATE bus\_customer\_tbl SET

Name = ?,

Email = ?,

Contact\_No = ?,

Passenger\_qty = ?,

Reserved\_date = ?,

Bus\_Type = ?,

Destination = ?,

Drivers = ?,

Status = ?

WHERE Customers\_Id = ?";

$stmt = $conn->prepare($sql);

$stmt->bind\_param(

"sssssssssi",

$name,

$email,

$contact\_no,

$passenger\_qty,

$reserved\_date,

$bus\_type,

$destination,

$drivers,

$status,

$customer\_id

);

if ($stmt->execute()) {

header('Location: ../pages/customers.php');

exit();

} else {

error\_log('Error updating record: ' . $stmt->error, 3, '../logs/errors.log');

header('Location: ../pages/error.php?error=UpdateFailed');

exit();

}

$stmt->close();

$conn->close();

}

?>

This code ensures better security, validation, and user feedback. Let me know if you need further assistance!

PAGES > BOOKING

Your PHP and HTML code for the booking calendar and sidebar menu appears to have the following features:

**Strengths**

1. **Dynamic Calendar Integration**:
   * The calendar dynamically displays the days for the selected month and year.
   * Booking indicators are added to calendar days based on data from the bus\_customer\_tbl.
2. **Responsive Sidebar Navigation**:
   * The sidebar includes menu options (Dashboard, Booking, Chats, Customers, Logout) with redirect functionality.
3. **Booking Details Modal**:
   * Clicking on a day with bookings shows details in a popup.
4. **Database Integration**:
   * Booking data is fetched from the database and encoded into a JSON format for use in JavaScript.
5. **Efficient Use of JavaScript**:
   * JavaScript handles navigation between months and dynamically generates the calendar.
6. **Session Management**:
   * Checks for active sessions before allowing access to the booking page.

**Improvements and Suggestions**

1. **SQL Injection Prevention**:
   * Use prepared statements for database queries to prevent SQL injection.  
     Example for your data fetch:
   * $query2 = $conn->prepare("SELECT \* FROM bus\_customer\_tbl");
   * $query2->execute();
   * $result2 = $query2->get\_result();
2. **Calendar Booking Range Logic**:
   * Currently, bookings are matched to a single day (Reserved\_date). If bookings span multiple days, the logic should reflect the range.
3. **Better Error Handling**:
   * Add error handling for database queries. Example:
   * if (!$result2) {
   * die("Error fetching data: " . $conn->error);
   * }
4. **User Experience Improvements**:
   * Highlight the current day in the calendar for better usability.
   * Allow filtering or searching bookings directly from the interface.
5. **Avoid Hardcoding Styles in HTML**:
   * Move inline styles to a separate CSS file to make the code cleaner and easier to maintain.
6. **Accessibility Enhancements**:
   * Add aria-label attributes for better screen reader compatibility.
7. **Duplicate Bookings JSON Declaration**:
   * You have already encoded the $bookings array into JSON in PHP. Avoid re-encoding it unnecessarily.
8. **Optimization for Large Data**:
   * If the bus\_customer\_tbl table grows, consider paginating results or limiting the data fetched.

**Code Cleanup Example**

Here's how a part of the script might look after applying some improvements:

**PHP (Fetching Data)**

require\_once('../pages/db\_connect.php');

$query = $conn->prepare("SELECT \* FROM bus\_customer\_tbl");

$query->execute();

$result = $query->get\_result();

$bookings = [];

while ($row = $result->fetch\_assoc()) {

$bookings[] = $row;

}

$bookingsJson = json\_encode($bookings);

**JavaScript (Improved Booking Indicator)**

const bookings = JSON.parse('<?php echo $bookingsJson; ?>');

function generateCalendar(month, year) {

const daysInMonth = new Date(year, month + 1, 0).getDate();

calendarBody.innerHTML = "";

for (let i = 1; i <= daysInMonth; i++) {

const dayCell = document.createElement("div");

dayCell.classList.add("day-cell");

const booking = bookings.find(b => {

const bookingDate = new Date(b.Reserved\_date);

return (

bookingDate.getDate() === i &&

bookingDate.getMonth() === month &&

bookingDate.getFullYear() === year

);

});

if (booking) {

dayCell.classList.add("has-booking");

dayCell.title = `${booking.Name}: ${booking.Bus\_Type}`;

dayCell.addEventListener("click", () => displayBookingDetails(booking));

}

dayCell.textContent = i;

calendarBody.appendChild(dayCell);

}

}

Let me know if you'd like me to refine or reorganize any specific part of your script further!

PAGES > CHATS

Your HTML structure and JavaScript functionality look solid for a simple chat page. Here are some observations and suggestions for improvement:

**Strengths:**

1. **Well-Structured Layout:**
   * The division between sidebar, chat sidebar, and chat main sections is clear and organized.
   * Responsive design considerations (e.g., flex usage) are well-implemented.
2. **User-Friendly Features:**
   * Functional menu with clear navigation.
   * Logout confirmation dialog is a good touch.
3. **Consistent Styling:**
   * The use of reusable classes like menu-item, chat-message, etc., ensures consistency.
4. **Accessibility:**
   * Good use of semantic HTML (e.g., h4, h5 tags) for better accessibility.

**Suggestions for Improvement:**

**1. Chat Sidebar Enhancements:**

* Add **hover effects** for chat message items in the sidebar to enhance user interactivity.

.chat-body .chat-message:hover {

background-color: #eaeaea;

cursor: pointer;

}

* Use a scrollbar style to improve appearance if the list grows large:

.chat-sidebar {

scrollbar-width: thin;

scrollbar-color: #ccc #f4f4f4;

}

**2. Message Input Enhancements:**

* Prevent empty message submission:

const sendButton = document.querySelector('.chat-footer button');

const messageInput = document.querySelector('.chat-footer input[type="text"]');

sendButton.addEventListener('click', function() {

if (messageInput.value.trim() === "") {

alert("Message cannot be empty!");

return;

}

// Append the message to chat-body or send via API

messageInput.value = ""; // Clear input

});

**3. Mobile Responsiveness:**

* Add media queries for mobile users:

@media (max-width: 768px) {

.chat-content {

flex-direction: column;

margin-left: 0;

}

.chat-sidebar {

flex: unset;

width: 100%;

}

.chat-main {

flex: unset;

width: 100%;

}

}

**4. Consistency in Admin Messages:**

* Ensure admin messages have proper alignment and styles:

.chat-message-admin {

text-align: right;

}

.chat-message-admin .chat-message-text {

align-self: flex-end;

background-color: #007bff;

color: white;

}

**5. Security and Best Practices:**

* **Sanitize user input** before rendering messages to prevent XSS attacks.
* Use **data attributes** for dynamically generated elements to avoid hardcoding IDs/classes.
* Load scripts at the end of the body for better performance.

Let me know if you'd like further refinements!

PAGES > CUSTOMERS

This PHP script represents a dynamic customer management page that fetches data from a database and displays it in a web interface, complete with CRUD (Create, Read, Update, Delete) functionality and user interaction. Here’s a breakdown of how it works:

**PHP Backend Logic**

1. **Session Management:**
   * session\_start();: Initializes a session to track user login state.
   * If $\_SESSION['id'] is not set (user is not logged in), the user is redirected to the login page (../index.php).
2. **Database Query:**
   * The script connects to the database via db\_connect.php.
   * It fetches all rows from the bus\_customer\_tbl table using a SELECT \* query.
   * Each row is appended to the $array variable using fetch\_assoc(), creating an array of associative arrays containing customer data.
3. **Data Output to Frontend:**
   * The $array variable is iterated using foreach in the HTML section to generate rows in an HTML table. Each row represents a customer with columns for their details.

**HTML and Bootstrap-Based Frontend**

1. **Sidebar and Navigation:**
   * A static sidebar includes links for navigation between pages (e.g., Dashboard, Booking, Customers, etc.).
   * Each link redirects to the respective page using window.location.href.
2. **Customer Table:**
   * A Bootstrap-styled table displays customer details (e.g., name, email, contact number).
   * Data is dynamically populated using PHP echo statements.
3. **Search Functionality:**
   * An input field allows users to search through the table rows.
   * A JavaScript event listener filters rows based on the search term, hiding non-matching rows.
4. **Modals for CRUD Operations:**
   * **Add Customer:** A modal form for adding a new customer. Submits data to ../functions/add\_customer.php.
   * **View Customer:** A modal displaying detailed information about a customer.
   * **Edit Customer:** A modal pre-filled with customer data. Submits changes to ../functions/edit\_customer.php.
   * **Delete Customer:** A confirmation modal for deleting a customer. Submits a deletion request to ../functions/delete\_customer.php.
5. **Bus Type Price Setting:**
   * A JavaScript function dynamically sets the price (Bus\_Price) based on the selected bus type (Bus or Coaster).
6. **Logout Button:**
   * Logs out the user by redirecting to logout.php?logout=true after confirmation.

**Key Features of the Output**

1. **Dynamic Customer Data Table:**
   * Displays data fetched from the database.
   * Enables real-time filtering with search functionality.
2. **Responsive UI:**
   * Uses Bootstrap for responsiveness, ensuring compatibility across devices.
3. **CRUD Operations:**
   * Full CRUD operations are supported through modals:
     + **Create:** Add new customer.
     + **Read:** View customer details.
     + **Update:** Edit customer information.
     + **Delete:** Remove customer records.
4. **Interactive Modals:**
   * Each modal is dynamically linked to a specific customer using their unique ID (Customers\_Id).
5. **Session Protection:**
   * Ensures only logged-in users can access this page.
6. **Integration with External Libraries:**
   * Bootstrap for styling.
   * JavaScript for interactivity.
   * AJAX and form submission for backend communication.

**Potential Outputs**

When the script is run:

* Logged-in users see a customer management page populated with data from bus\_customer\_tbl.
* Users can interact with the table to add, edit, or delete customers.
* Search functionality filters customers in real time.
* Modals allow for clean and organized data management.
* Selecting a bus type automatically updates its price in the form.

Let me know if you'd like help with a specific part of the code or functionality!

PAGES > DB\_CONNECT

This code connects your PHP application to a MySQL database.

1. **Database Information:**
   * $servername: The address of your database (with port 3306).
   * $username: Your database username.
   * $password: Your database password.
   * $dbname: The name of your database.
2. **Create Connection:**
   * new mysqli(...) tries to connect to the database using the information above.
3. **Check for Errors:**
   * If the connection fails, it stops the script and shows an error with:  
     "Connection failed: [error details]".

**In short:**

* If all the details are correct, you connect to the database successfully.
* If not, it will show an error.

Let me know if you want a super simplified example or help!

In traffic congestion, the roadways are crowded with an abundance of vehicles causing delays and inefficiencies in travel. The Online Bus Ticket Booking System reduces the need for people to visit bus terminals and hence decreases vehicular trips and alleviates road congestion.