3rdiTech Leave Application Portal Documentation

ANJALI VAISH

Step-by-Step Documentation:

<mark>Creating a Custom Login Page</mark>
1. Initialized HTML Structure

- Created a basic HTML structure for the login page.
- Included necessary meta tags and links to external CSS and JavaScript files.
- Set up the main container (`<div id="main">`) to hold all the content

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Page</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="main">
        <!-- Login form and other content will be added here -->
    </div>
    <script src="script.js"></script>
</body>
</html>
```

2. Applied Basic CSS Styling

- Applied initial styling in CSS to reset default browser styles.
- Set up the basic layout structure for the login page

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    font-family: Verdana, Geneva, Tahoma, sans-serif;
    color: #fff;
}
html,body {
    height: 100%;
    width: 100%;
}
#main {
    position: relative;
    height: 100%;
}
```

3. Added Background Image

- Selected a suitable background image for the login page.
- Incorporated the background image using CSS within the #content container.

```
#content img {
    height: 100vh;
    width: 100vw;
    object-fit: cover;
    position: absolute;
    z-index: -1;
    top: 0;
    left: 0;
}
```

4. Created the Login Form

- Designed the login form structure within the `#login` container.
- Included fields for email, password, and a submit button

```
<div id="login">
    <h2>Login</h2>
    <form>
        <label for="email">Email ID:</label><br>
        <input type="email" id="email" name="email"><br>
        <label for="password">Password:</label><br>
        <input type="password" id="password" name="password"><br><br>
        <input type="submit" value="Login">
    </form>
</div>
```

5. Styled the Login Form

- Applied CSS to style the login form, including fonts, colors, and layout.
- Customized the appearance of form elements like inputs and buttons.

```
#login {
    position: fixed;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    background-color: #3fb4f8;
    padding: 20px;
    border-radius: 15px;
    z-index: 99;
    color: black;
    /* Additional styling properties */
}
```

6. Added Login Options

- Included options for admin and employee login using icons or buttons.
- Applied appropriate CSS to style the login options.

```
<div class="login-options">
    <div class="login-option" id="admin-login">
        <!-- Admin login icon and text -->
    </div>
    <div class="login-option" id="employee-login">
        <!-- Employee login icon and text -->
    </div>
</div>
```

Workflow Explanation:

- The HTML structure was set up first, followed by gradual application of CSS styling to create the desired layout and appearance.
- Login options and additional features were added step by step, ensuring seamless integration with the existing design.
- Testing was conducted throughout the process to identify and address any issues promptly.
- The workflow followed an iterative approach, allowing for adjustments and refinements based on testing results and user feedback.

Creating a user dashboard

Introduction

This document provides a detailed step-by-step guide for creating a user dashboard web application. The project involved setting up HTML and CSS, and integrating external resources. It also includes the steps taken to troubleshoot errors and enhance the functionality of the dashboard.

Initial setup

1.Created Project Structure:
- Created index.html for the main HTML structure.
- Created style1.css for styling.
- Created script1.js for potential JavaScript functionalities (though not utilized in the steps outlined).

HTML Structure

1.Set Up Basic HTML Skeleton:

- Defined the <!DOCTYPE html> and the html, head, and body tags.

- Linked external CSS file style1.css.
- Included the Remixicon library for icons via a CDN link:

```
href="https://cdn.jsdelivr.net/npm/remixicon/fonts/remixicon.css">
```

3.Added Main HTML Content:

- Created a div with id="main" to serve as the container for all content.
- Added a navigation bar (nav1) with a logo, user dashboard text, notification icon, and logout button.
- Set up a secondary navigation (nav2) with buttons for Dashboard, Leave Calendar, Leave Status, and a settings icon.
- Added page1-inner to display welcome text and user profile information.
- Included page2 to add additional content.

CSS Styling

1.Global Styles:

- Applied global styles to reset margin, padding, and box-sizing.
- Set `html` and `body` to 100% height and width, and disabled overflow

```
html, body {
    height: 100%;
    width: 100%;
    overflow: hidden;
}
```

2.Styled Navigation Bars:

- Styled nav1 for the top navigation bar with a black background and flexbox for alignment.
- Styled nav2 for the side navigation with a dark background and vertical alignment.

3.Styled Page Content:

- Configured page1-inner to use flexbox for layout and divided it into two equal parts (inner-left and inner-right).
- Applied a gradient background to inner-left and set a white background for inner-right.

4.Ensured Responsive Design:

- Used percentages and flexible units for dimensions to ensure the layout adapts to different screen sizes.

Extended Page Content

1.Extended page2 Length:

- Increased the height of page2 to 300% to allow for additional scrolling content.

Troubleshooting and Adjustments

1.Fixed Positioning Issues:

- Adjusted the positions of nav1 and nav2 to be fixed so they remain in place while scrolling.
- Set page1-inner to fixed positioning and added an overflow property to enable scrolling within the element:

```
#page1-inner {
    position: fixed;
    top: 20%;
    left: 20%;
    width: 80%;
    height: calc(100% - 20%);
    overflow-y: auto;
    display: flex;
}
```

2.Added Content to page2:

- Included additional paragraphs in page2 to test scrolling functionality.

External Resources:

Remixicon: Used for icons in the navigation bars. Linked via:

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/remixicon/fonts/remixicon.css">

Workflow Explanation:

HTML and CSS Setup:

- Started by setting up the HTML structure to organize the content logically.
- Styled the elements using CSS to create a visually appealing layout.

Functionality Enhancements:

- Adjusted positions and dimensions to ensure the navigation bars remained fixed.

- Enabled scrolling within page1-inner to manage content overflow.

Testing and Refinement:

- Tested the layout on different screen sizes to ensure responsiveness.

- Added content to page2 to confirm the scrolling functionality works as intended.

Conclusion:

This guide provides a comprehensive overview of creating a user dashboard, including setting up HTML and CSS, adjusting positions for fixed navigation, and enhancing content display. Following these steps will help in creating a responsive and functional user dashboard.

Creating a Multi-Page Dashboard with Navigation Bars

Introduction:

This documentation provides a detailed guide on creating a multi-page dashboard with consistent navigation bars using HTML and CSS. The process includes creating HTML structure, styling with CSS, and troubleshooting common issues.

1. Set Up the HTML Structure:

- Created the HTML file: Created an HTML file named index.html

2. Apply CSS for Styling:

- Created the CSS file: Created a CSS file named style1.css with the initial styling for the HTML structure.

Troubleshooting and Adjustments:

1.Troubleshot visibility issues:

- Adjusted the positioning and height properties to ensure both pages were displayed correctly.
- Initially, #page1 was not displayed due to incorrect positioning. Adjusted #page1 from absolute to relative.
- Ensured #page1 and #page2 were displayed one after another by maintaining their height and width.

2.Final adjustments:

- Verified and corrected the layout and ensured navigation bars were consistent across both pages.

Workflow Explanation:

- HTML Structure: Defined the layout with two main sections, #page1 and #page2, each with its own content.
- CSS Styling: Applied CSS styles to define the appearance, including background colors, text styles, and layout positioning.
- Debugging: Fixed issues with element visibility by adjusting CSS properties like position, height, and width.

Conclusion:

By following these steps, we successfully created a multi-page dashboard with consistent navigation bars and ensured both pages were displayed correctly. This process involved setting up HTML, styling with CSS, and troubleshooting common layout issues.

Real-Time Employee Data Dashboard

Introduction:

This documentation outlines the steps taken to create a real-time employee data dashboard, including the integration of employee data from a CSV file, the use of various tools for data processing and visualization, and the troubleshooting of issues encountered along the way.

Tools and Libraries Used

- Python (Pandas, Plotly, Dash, Dash Bootstrap Components)
- Jupyter Notebook
- Plotly Dash Documentation: Plotly Dash
- Dash Bootstrap Components Documentation: Dash Bootstrap Components

1. Set Up the Environment

- Imported necessary libraries for data manipulation and visualization.
- Imported Dash and Dash Bootstrap Components for web app creation.

```python
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from dash import Dash, dcc, html, Input, Output
import dash_bootstrap_components as dbc
```

2. Load and Process the Data

- Loaded the employee data from a CSV file.
- Checked and cleaned the data as necessary.

```python
df = pd.read_csv('path/to/employee_data.csv')
df.dropna(inplace=True)  # Dropped any rows with missing values
df['start_date'] = pd.to_datetime(df['start_date'])  # Converted start_date to datetime
```

3. Initialize the Dash App

- Initialized the Dash app and applied a Bootstrap theme for styling.

app = Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])

4. Create Visualization Components

- Created different plots and graphs to visualize the data.
- Employee distribution by department.
- Average salary by department.
- Employee distribution by start date.

```python
# Employee distribution by department
fig_dept_dist = px.bar(df, x='department', title='Employee Distribution by Department')

# Average salary by department
fig_avg_salary = px.bar(df.groupby('department')['salary'].mean().reset_index(),
                        x='department', y='salary', title='Average Salary by Department')

# Employee distribution by start date
fig_start_date = px.histogram(df, x='start_date', title='Employee Distribution by Start Da
```

5. Design the Layout

- Designed the layout of the dashboard using Dash's HTML and Core Components.
- Used Bootstrap components to enhance the appearance and responsiveness.

```
app.layout = dbc.Container([
    dbc.Row(dbc.Col(html.H1("Employee Data Dashboard"), className="mb-2")),
    dbc.Row([
        dbc.Col(dcc.Graph(figure=fig_dept_dist), md=4),
        dbc.Col(dcc.Graph(figure=fig_avg_salary), md=4),
        dbc.Col(dcc.Graph(figure=fig_start_date), md=4)
    ]),
    dbc.Row(dbc.Col(html.H5("Data Updated: " + pd.Timestamp.now().strftime('%Y-%m-%d %H:%M
])
```

6. Define Callbacks (If Any)

- Implemented callbacks to make the dashboard interactive (if applicable).

```
@app.callback(
    Output('output-component-id', 'children'),
    [Input('input-component-id', 'value')]
)
def update_output(value):
    return f'You have entered {value}'
```

7. Run the App

- Set the app to run on a specified port and debug mode for development purposes.

Issues Encountered and Troubleshooting

- Issue: Inconsistent date formats
  Solution: Ensured all date columns were converted to a consistent datetime format using pd.to_datetime().
- Issue: Missing values in the dataset
  Solution: Used df.dropna() to remove rows with missing values.

- Issue: Layout not responsive
  Solution: Utilized Dash Bootstrap Components to create a responsive grid layout.

Conclusion

The real-time employee data dashboard was successfully created using Dash and Plotly. It provided visual insights into employee distribution, salary averages, and start date trends. Issues encountered were resolved by ensuring consistent data formatting and using responsive design principles. The dashboard can be easily extended or modified to include additional data visualizations or functionalities as required.

Creating an Interactive Calendar with Evo Calendar

Introduction

This documentation outlines the process of creating an interactive calendar using the Evo Calendar plugin. It covers the setup of HTML, CSS, and JavaScript files, as well as the configuration of calendar events. The step-by-step instructions, references to external resources, and a detailed workflow explanation are provided to ensure a smooth implementation.

1. Setup Project Structure:

- Created a new project directory.
- Created three files within the project directory:
  - ➔ index.html (HTML file)
  - ➔ style2.css (CSS file)
  - ➔ evo-calendar.min.css and evo-calendar.midnight-blue.min.css (CSS files for Evo Calendar themes)
  - ➔ evo-calendar.min.js (JavaScript file for Evo Calendar functionality)

2. Included External Libraries:

- Included jQuery and Evo Calendar library via the CDN in the index.html file

```
<script src="https://cdn.jsdelivr.net/npm/jquery@3.4.1/dist/jquery.min.js"></script>
<script src="evo-calendar.min.js"></script>
```

- Linked Google Fonts for custom fonts in the index.html file.

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Montserrat:ital,wght@0,100..900;1
```

3. HTML Structure

- Created the HTML structure in index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Calendar</title>
    <link rel="stylesheet" href="style2.css">
    <link rel="stylesheet" href="evo-calendar.midnight-blue.min.css">
</head>
<body>
    <div class="hero">
        <div id="calendar"></div>
    </div>
</body>
</html>
```

4. Applied CSS for Styling

- Created style2.css and added custom styles

```css
body {
    font-family: 'Montserrat', sans-serif;
}


.hero {
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background-color: #f5f5f5;
}


#calendar {
    width: 80%;
}
```

5. Initialized Evo Calendar

- ● Initialized the Evo Calendar in the index.html by adding the following script

```html
<script>
    $(document).ready(function(){
        $('#calendar').evoCalendar({
            calendarEvents: [
                {
                    id: 'event1',
                    name: "New Year",
                    date: "January/1/2024",
                    description: "Wish everyone a prosperous and happy new year!",
                    type: "holiday",
                    everyYear: true
                },
                // Additional events here
            ],
            todayHighlight: true,
            sidebarToggler: true,
            format: 'mm/dd/yyyy'
        });
    });
</script>
```

6. Configured Calendar Events

- Defined multiple calendar events for the year 2024 within the Evo Calendar initialization script

```
calendarEvents: [
    {
        id: 'event1',
        name: "New Year",
        date: "January/1/2024",
        description: "Wish everyone a prosperous and happy new year!",
        type: "holiday",
        everyYear: true
    },
    {
        id: 'event2',
        name: "Republic Day",
        date: "January/26/2024",
        description: "Celebrating the adoption of the Indian Constitution.",
        type: "holiday",
        everyYear: true
    },
    // Additional events here
]
```

External Resources Used

- jQuery CDN: https://cdn.jsdelivr.net/npm/jquery@3.4.1/dist/jquery.min.js
- Evo Calendar Library: https://github.com/edlynvillegas/evo-calendar
- Google Fonts: https://fonts.googleapis.com

Workflow Explanation

- Initiated by creating the project structure and necessary files.
- Incorporated external libraries and Google Fonts into the HTML file for additional functionalities and styling.
- Developed the HTML structure to host the calendar.
- Styled the project using a custom CSS file to enhance visual appearance.
- Initialized the Evo Calendar and configured multiple events to display on the calendar.
- Ensured all elements were functioning correctly by testing in a web browser.

This documentation detailed the process of creating an interactive calendar using the Evo Calendar plugin. By following the step-by-step instructions, setting up the project structure, and incorporating necessary libraries, a functional and visually appealing calendar was successfully implemented. The documentation provided guidance on configuring calendar events, applying custom styles, and ensuring all elements functioned correctly. Utilizing external resources like jQuery and Evo Calendar streamlined the implementation, enabling a robust and customizable calendar solution.

Creating a Leave Requests Web Page

Introduction

This documentation provides a step-by-step guide to creating a leave requests web page with filtering options and status summaries. The goal was to build a functional and visually appealing interface for managing leave requests, including features for adding, editing, and deleting leave types. The process involved creating HTML structure, applying CSS for styling, and ensuring a user-friendly layout.

1. Setup HTML Structure

- Created the HTML Document
- Defined the basic structure of the HTML document with the necessary metadata.
- Added a title "Requests" and linked an external stylesheet (style8.css).

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Requests</title>
    <link rel="stylesheet" href="style8.css">
</head>
<body>
    <div class="container">
        <h1>Leave Requests</h1>
        <!-- Status boxes and table will be added here -->
    </div>
</body>
</html>
```

2. Added Status Summary Boxes:

- Included three boxes to display the number of leave requests that are pending, approved, and denied.
- Used flexbox to arrange them horizontally.

```html
<div class="status-boxes">
    <div class="status-box pending">
        <h2>Pending</h2>
        <p class="count">0</p>
    </div>
    <div class="status-box approved">
        <h2>Approved</h2>
        <p class="count">0</p>
    </div>
    <div class="status-box denied">
        <h2>Denied</h2>
        <p class="count">0</p>
    </div>
</div>
```

3. Included Filtering Options:

- Added dropdown and date input for filtering leave requests by status and date.

```html
<div class="filter-options">
    <label for="status-filter">Filter by Status:</label>
    <select id="status-filter">
        <option value="all">All</option>
        <option value="pending">Pending</option>
        <option value="approved">Approved</option>
        <option value="denied">Denied</option>
    </select>
    <label for="date-filter">Filter by Date:</label>
    <input type="date" id="date-filter">
</div>
```

- Created a table to display leave requests with columns for employee name, leave type, start and end dates, status, and actions.
- Included buttons for approving or denying requests within the actions column.

```
<table>
    <thead>
        <tr>
            <th>Employee Name</th>
            <th>Leave Type</th>
            <th>Start Date</th>
            <th>End Date</th>
            <th>Status</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
        <!-- Table rows will be added dynamically -->
    </tbody>
</table>
```

5. Applied CSS for Styling:

- Added CSS rules to style the containers, status boxes, filter options, and table.
- Used different shades of transparent colors for the status boxes and applied hover effects.
- Set the background color of the table to light yellow.

Conclusion

The process of creating the leave requests web page involved defining the HTML structure, applying CSS for styling, and ensuring a functional user interface. The final product includes a visually appealing layout with status summary boxes, filtering options, and a table for managing leave requests. The design focused on user experience, providing clear and intuitive controls for administrators to manage leave requests effectively.

Creating a Multi-Page Login and Dashboard Application

Introduction

This documentation outlines the step-by-step process of creating a multi-page login and dashboard application. The application includes two main functionalities: admin login leading to an admin dashboard and user login leading to a user-specific page. The admin dashboard features navigation to additional pages for leave types and leave requests. The entire process is described from setting up the HTML structure to adding JavaScript functionalities for navigation.

1. Setup Project Structure:

- Created the project directory and added the following files:
➜ Index.html
➜ Index6.html
➜ Index7.html
➜ Index8.html
➜ Style.css
➜ Style6.css
➜ script.js
➜ script6.js

2. Index Page Creation (index.html):

- Set up the basic HTML structure for the login page with buttons for admin and user login.
- Applied CSS to style the login page (style.css).
- Added a script reference to script.js with the defer attribute to ensure proper loading.

3. JavaScript for Login Page (script.js):

- Added event listeners for the admin and user login buttons to highlight the selected button.

● Implemented form submission handling to redirect to the respective dashboard pages based on the selected login type.

```javascript
document.getElementById("admin-login").addEventListener("click", function() {
    selected = "admin";
    selectButton(this);
});

document.getElementById("employee-login").addEventListener("click", function() {
    selected = "user";
    selectButton(this);
});
```

```javascript
document.getElementById("login-form").addEventListener("submit", function(event) {
    event.preventDefault();
    if (selected === "user") {
        window.location.href = "index1.html";
    } else if (selected === "admin") {
        window.location.href = "index6.html";
    } else {
        alert("Please select a login option.");
    }
});

function selectButton(button) {
    document.querySelectorAll(".login-option").forEach(btn => {
        btn.classList.remove("selected");
    });
    button.classList.add("selected");
}
```

4. Admin Dashboard Page Creation (index6.html):

● Created the HTML structure for the admin dashboard including a navigation bar and a chart container.
● Applied CSS for the admin dashboard styling (style6.css).

- Added references to external libraries like Remixicon and Chart.js for icons and chart functionalities.

5. JavaScript for Admin Dashboard (script6.js):

- Used Chart.js to create a bar chart displaying leaves taken each month.
- Highlighted specific months (April, October, November) with different colors.

6. Linking Additional Pages to Admin Dashboard:

- Created index7.html for Leave Types and index8.html for Leave Requests with appropriate HTML structure and styling.
- Added event listeners to buttons in index6.html to redirect to these pages.

```javascript
document.getElementById("leave-types-btn").addEventListener("click", function() {
    window.location.href = "index7.html";
});

document.getElementById("leave-requests-btn").addEventListener("click", function() {
    window.location.href = "index8.html";
});
```

7. Logout Functionality:

- Added an event listener to the "Logout" button in index6.html to redirect back to index.html.

```javascript
document.getElementById("logout-btn").addEventListener("click", function() {
    window.location.href = "index.html";
});
```

External References

- Used icons from Remixicon

- Utilized Chart.js for charting functionalities
- Included Font Awesome for additional icons

Errors and Debugging

- Encountered issues with JavaScript event listeners not firing due to incorrect element IDs. Ensured all IDs matched between HTML and JavaScript.
- Addressed Chart.js rendering issues by ensuring the canvas element was correctly selected and the Chart.js library was properly loaded.

Conclusion

This documentation detailed the creation of a multi-page login and dashboard application with admin-specific functionalities. The application dynamically navigates between different pages based on user interactions, demonstrating a practical use of HTML, CSS, and JavaScript for web development. The steps included setting up the project structure, implementing page redirection, and enhancing user experience with interactive elements.

Dynamic Leave Request System

Introduction

This documentation provides a comprehensive guide to creating a dynamic leave request system using HTML, CSS, JavaScript, PHP, and MySQL. The system includes a user interface for employees to apply for leave and an admin interface to manage leave requests. The instructions cover setting up the environment, creating necessary files, implementing dynamic data handling, and troubleshooting common issues.

1. Setting Up the Project:

- Created a folder named dv2js in the htdocs directory of XAMPP.
- Within the dv2js folder, created the following files: index8.html, admin_dashboard.php, script8.js, style8.css, fetch_leave_requests.php, and db_connection.php.

```
<!-- index8.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Requests</title>
    <link rel="stylesheet" href="style8.css">
</head>
<body>
    <!-- HTML structure for admin dashboard -->
</body>
</html>
```

2. HTML Structure for Admin Dashboard (index8.html):

- Created the HTML structure for the admin dashboard, including elements for displaying leave requests in a table format.
- Added placeholders for displaying pending, approved, and denied leave requests.
- Included filter options for status and date filtering.

```
<!-- HTML structure for admin dashboard -->
<div class="container">
    <!-- Leave request display area -->
    <table id="leave-requests">
        <!-- Table headers -->
        <thead>
            <!-- Table rows will be added dynamically -->
        </thead>
        <!-- Table body -->
        <tbody>
            <!-- Table rows will be added dynamically -->
        </tbody>
    </table>
</div>
```

3. CSS Styling (style8.css):

- Applied CSS styling to customize the appearance of the admin dashboard, including styling for table elements, filter options, and status boxes.
- Used CSS to ensure responsive design and consistent layout across different screen sizes.

```css
/* CSS styling for admin dashboard */
.container {
    /* Styles for container */
}
/* Other styles for table, filter options, etc. */
```

4. JavaScript for Fetching Leave Requests (script8.js):

- Implemented JavaScript functions to fetch leave requests from the server using AJAX.
- Populated the table with leave request data dynamically by appending table rows for each request.
- Included event listeners for filter options and buttons for approving or denying leave requests.

```javascript
// JavaScript for fetching leave requests
function fetchLeaveRequests() {
    // Fetch leave requests using AJAX
}
// Event listener for loading page
window.addEventListener('load', fetchLeaveRequests);
```

5. PHP Script for Fetching Leave Requests (fetch_leave_requests.php):

- Created a PHP script to handle database interaction and fetch leave requests from the leave_requests table.
- Used PDO (PHP Data Objects) to establish a database connection and execute SQL queries.
- Returned leave request data as JSON to be consumed by the JavaScript function.

```php
<?php
// PHP script for fetching leave requests
// Include database connection
include 'db_connection.php';


// Fetch leave requests from the database
$stmt = $connection->prepare("SELECT * FROM leave_requests");
$stmt->execute();
$leaveRequests = $stmt->fetchAll(PDO::FETCH_ASSOC);


// Return leave requests data as JSON
header('Content-Type: application/json');
echo json_encode($leaveRequests);
?>
```

6. Database Connection (db_connection.php):

- Configured the PHP script to establish a connection to the MySQL database using PDO.
- Specified database credentials such as hostname, username, password, and database name.
- Encapsulated database connection logic to ensure reusability across multiple PHP scripts.

```php
<?php
// PHP script for database connection
try {
    $connection = new PDO('mysql:host=localhost;dbname=your_database', 'username', 'pas
} catch (PDOException $e) {
    echo "Connection failed: " . $e->getMessage();
    exit();
}
?>
```

7. Integrating Admin Dashboard (admin_dashboard.php):

- Implemented PHP logic to authenticate users and restrict access to the admin dashboard only for authenticated admins.
- Included index8.html within the admin_dashboard.php file using PHP include directive to display the admin dashboard.

```php
<?php
// PHP logic for authenticating users and restricting access
session_start();
if (!isset($_SESSION['user_id']) || $_SESSION['user_role'] !== 'admin') {
    // Redirect unauthorized users
    header("Location: login.php");
    exit();
}
?>
<!-- Include index8.html -->
<?php include 'index8.html'; ?>
```

8. Testing and Error Resolution:

- Tested the admin dashboard by accessing admin_dashboard.php in the browser.
- Encountered errors such as incorrect data display or database connection issues.

- Resolved errors by debugging PHP scripts, checking database configurations, and verifying SQL queries.

Conclusion

In conclusion, we have successfully created an admin dashboard for managing leave requests. The dashboard dynamically fetches leave requests from the database, displays them in a table format, and allows admins to take actions on requests. By following the steps outlined in this documentation and addressing any encountered errors, we have built a functional admin dashboard that enhances the leave request management process.

Backend and Frontend Integration

Introduction

This documentation provides a detailed, step-by-step guide to creating and integrating a leave portal application with both user and admin functionalities. The guide includes backend and frontend implementation, error handling, and dynamic status updates. The goal was to ensure that leave requests submitted by users are dynamically displayed and managed by the admin, with real-time status updates reflected in the user interface.

Backend Setup

1. Created Database and Table

- Created a MySQL database named `leave_application_db`.
- Created a table `leave_requests` with the following schema:

```sql
CREATE TABLE leave_requests (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    leave_type VARCHAR(50),
    start_date DATE,
    end_date DATE,
    status VARCHAR(50) NOT NULL,
    reason TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

PHP Script to Submit Leave Requests

1. Created `submit_leave_request.php

- This script handled the submission of leave requests from the user form and stored the data in the database.

Fetch Leave Requests for Admin

1. Created `fetch_leave_requests.php

- This script fetched all leave requests from the database to be displayed on the admin dashboard.

```php
<?php
include 'db_connection.php';


$stmt = $connection->query("SELECT * FROM leave_requests");
$leaveRequests = $stmt->fetchAll(PDO::FETCH_ASSOC);


header('Content-Type: application/json');
echo json_encode($leaveRequests);
?>
```

Update Leave Request Status

1. Created `update_status.php

- This script updated the status of leave requests based on admin actions.

```php
<?php
include 'db_connection.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $stmt = $connection->prepare("UPDATE leave_requests SET status=? WHERE id=?");
    $stmt->execute([$_POST['status'], $_POST['id']]);

    if ($stmt->rowCount() > 0) {
        echo "success";
    } else {
        echo "Error: " . $connection->errorInfo()[2];
    }
} else {
    echo "This script can only be accessed via a POST request.";
}
?>
```

Admin Dashboard

Admin Dashboard

1. Created `admin_dashboard.php'

- Integrated `index8.html` to display leave requests dynamically and handle status updates.

2. Updated `index8.html`

- Styled the approve and decline buttons and added functionality to highlight the status column.

3. Created `script8.js`

- Added JavaScript to fetch leave requests and update the status dynamically.

User Dashboard

1. Created `index3.html`

- Designed a user dashboard to display the status of their leave requests.

2. Created `script3.js`

- Added JavaScript to fetch and display leave requests for the logged-in user dynamically.

Ensure Correct Data in Database

1. Verified Data Integrity

- Ensured the `status` field in the database correctly stored the status of each leave request (e.g., 'approved', 'pending', 'declined').

Errors Encountered and Resolutions

- Error: Status field was blank in the user dashboard.
  Resolution: Ensured the `fetch_user_leave_requests.php` script correctly fetched the status and included it in the JSON response. Added logging in JavaScript to debug data fetching.

```javascript
console.log(data); // Debug: Log fetched data
```

- Error: Admin actions not reflecting in the user dashboard.
  Resolution: Added highlighting to the status column and ensured dynamic updates were handled in `script8.js`.

```javascript
const statusCell = button.closest('tr').querySelector('.status-cell');
statusCell.textContent = status;
if (status === 'approved') {
    statusCell.classList.add('highlight-approved');
    statusCell.classList.remove('highlight-declined');
} else if (status === 'denied') {
    statusCell.classList.add('highlight-declined');
    statusCell.classList.remove('highlight-approved');
}
```

Conclusion

This documentation provided a comprehensive guide to integrating backend and frontend functionalities for a leave portal application. The guide covered setting up the database, creating necessary PHP scripts, and dynamically updating the user and admin dashboards. Errors encountered during development were resolved through debugging and code adjustments, ensuring the application worked as intended.

Introduction

This documentation provides a comprehensive step-by-step guide to creating a leave portal application where users can apply for leave and check the status of their requests. It also covers troubleshooting steps for common issues encountered during development.

Setup the Database:

1.Created the Database and Table:

- Used phpMyAdmin to create a database named leave_portal.
- Created a table named leave_requests with the following columns:
➔ id (INT, AUTO_INCREMENT, PRIMARY KEY)
➔ user_id (INT)
➔ leave_type (VARCHAR)
➔ start_date (DATE)
➔ end_date (DATE)
➔ reason (TEXT)
➔ status (VARCHAR)
- SQL Query to create the table:

```sql
CREATE TABLE leave_requests (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    leave_type VARCHAR(255),
    start_date DATE,
    end_date DATE,
    reason TEXT,
    status VARCHAR(255)
);
```

<u>Setup the Frontend:</u>

Created index3.html:

- Designed the web page layout using HTML.
- Applied CSS for styling using style3.css.
- Included a script file script3.js for dynamic content rendering.

<u>Setup the Backend:</u>

Created fetch_user_leave_requests.php:

- Fetched leave requests for the logged-in user from the database.
- Returned the data as a JSON response.
- PHP Code:

```php
<?php
session_start();
include 'db_connection.php';

// Assuming user_id is stored in session after login
$user_id = $_SESSION['user_id'];

try {
    $stmt = $connection->prepare("SELECT leave_type, start_date, end_date, statu
    $stmt->execute([$user_id]);
    $leaveRequests = $stmt->fetchAll(PDO::FETCH_ASSOC);

    header('Content-Type: application/json');
    echo json_encode($leaveRequests);
} catch (Exception $e) {
    error_log("Error fetching leave requests: " . $e->getMessage());
    echo json_encode([]);
}
?>
```

<u>JavaScript for Dynamic Content:</u>

<u>Created script3.js:</u>

- Fetched leave requests from the backend and displayed them on the user interface.
- Provided functionality to filter and sort leave requests.

<u>Troubleshooting and Error Resolution</u>

<u>Issue: No Status Display on User Side-</u>

- Error Logs: Checked the PHP error logs and found an error indicating an unknown column 'user_id' in the SQL query.
  Resolution: Added the user_id column to the leave_requests table in the database and ensured it was included in the session and query.

<u>Issue: Form Submission Not Storing Data-</u>

- Diagnosis: Checked network requests in the browser developer tools and found that the data was not being sent correctly.
  Resolution: Verified the PHP script handling the form submission, ensured the user_id was being correctly retrieved from the session, and the SQL query included all necessary fields.

<u>Issue: Missing Required URL-</u>

- Error: "Not Found The requested URL was not found on this server."
  Resolution: Verified the file paths and ensured all files were correctly placed in the server directory.

<u>Conclusion</u>
Creating a leave portal application involves setting up the database, designing the frontend, implementing backend logic, and handling dynamic content with JavaScript. Troubleshooting common issues requires careful inspection of error logs, verifying database structure, and ensuring proper data flow between frontend and backend

components. Following these steps ensures a functional and user-friendly leave management system.

Introduction

This documentation outlines the step-by-step process for creating a dynamic leave management system that updates leave balances automatically when a leave request is submitted and approved. The system comprises two main parts: the user-side application for submitting leave requests and viewing leave balances, and the admin-side application for managing leave requests.

Setting Up the Environment

- Installed XAMPP/WAMP to set up a local development environment with Apache, MySQL, and PHP.
- Created a MySQL Database named leave_application_db.
- Created Two Tables: users and leave_requests.

Creating the User-Side Application

1. Apply for Leave Page (index5.html)

- Created an HTML Form to submit leave requests.
- Replaced the Name Field with an Email field for user identification.
- Implemented JavaScript to Handle Form Submission

2. Submit Leave Request Script (submit_leave_request.php)

- Handled Form Data and Inserted it into the Database.
- Ensured Unique Constraint on Email to prevent duplicate entries.

```php
<?php
include 'db_connection.php';
$email = $_POST['email'];
$start_date = $_POST['start_date'];
$end_date = $_POST['end_date'];
$leave_type = $_POST['leave_type'];
$reason = $_POST['reason'];


$stmt = $connection->prepare("INSERT INTO leave_requests (email, start_date, end_date
$stmt->bind_param("sssss", $email, $start_date, $end_date, $leave_type, $reason);
$stmt->execute();
echo json_encode(['status' => 'success', 'message' => 'Leave request submitted succes
?>
```

Creating the Admin-Side Application

1. Admin Leave Requests Page (index.html)

- Displayed Leave Requests in a table format.
- Implemented Approve and Decline Buttons to manage leave requests.
- Updated the HTML and JavaScript to include email and reason fields

2. Fetch Leave Requests Script (fetch_leave_requests.php)

- Retrieved Leave Requests from the Database and returned them as JSON

```php
<?php
include 'db_connection.php';
$stmt = $connection->query("SELECT * FROM leave_requests");
$leaveRequests = $stmt->fetchAll(PDO::FETCH_ASSOC);
header('Content-Type: application/json');
echo json_encode($leaveRequests);
?>
```

3. Update Status Script (update_status.php)

- Handled Status Updates for leave requests and adjusted leave balances

```php
<?php
include 'db_connection.php';
$status = $_POST['status'];
$id = $_POST['id'];
$stmt = $connection->prepare("UPDATE leave_requests SET status=? WHERE id=?");
$stmt->bind_param("si", $status, $id);
if ($stmt->execute()) {
    echo "success";
} else {
    echo "Error: " . $stmt->error;
}
?>
```

Displaying Leave Balances

1. Leave Balance Summary Page (index4.html)

- Displayed Various Leave Types and Balances for the logged-in user.
- Removed Paternity Leave Option and adjusted the design

2. Fetch User Leave Balances (fetch_user_leave_balances.php)

- Retrieved and Calculated Leave Balances based on approved leave requests

```php
<?php
include 'db_connection.php';
$email = $_SESSION['email']; // Assuming user email is stored in session
$stmt = $connection->prepare("SELECT leave_type, SUM(DATEDIFF(end_date, start_date)
$stmt->bind_param("s", $email);
$stmt->execute();
$result = $stmt->get_result();
$leave_balances = [];
while ($row = $result->fetch_assoc()) {
    $leave_balances[$row['leave_type']] = $row['used_days'];
}
echo json_encode($leave_balances);
?>
```

3. Update Leave Balance Summary Dynamically (script4.js)

- Updated the Leave Balance Summary on the user interface

```javascript
document.addEventListener('DOMContentLoaded', function() {
    fetchLeaveBalances();
});

function fetchLeaveBalances() {
    fetch('fetch_user_leave_balances.php')
        .then(response => response.json())
        .then(data => {
            document.getElementById('annual-leave').textContent = `${data.annual || 0}
            document.getElementById('sick-leave').textContent = `${data.sick || 0}/3 d
            document.getElementById('maternity-leave').textContent = `${data.maternity
            document.getElementById('family-leave').textContent = `${data.family || 0}
            document.getElementById('bereavement-leave').textContent = `${data.bereave
        })
        .catch(error => console.error('Error fetching leave balances:', error));
}
```

Errors Encountered and Resolutions

- Duplicate Entry Error: Encountered a Duplicate entry error due to the unique
  constraint on the email field in the leave_requests table.
  Resolution: Ensured unique email addresses by adding appropriate validation
  before inserting new records.
- Decline Button Not Functioning: The decline button did not update the status
  correctly.
  Resolution: Corrected the JavaScript and PHP code handling the decline status
  to ensure it updates the database and user interface correctly.
- Filter Options Not Working: The custom filter option was removed as it did not
  function as expected.
  Resolution: Simplified the filter options and ensured they functioned as described
  (Earliest, Latest, Approved).

Conclusion

This documentation provides a detailed step-by-step guide to creating a dynamic leave management system. The system allows users to submit leave requests, view their leave balances, and for admins to manage leave requests efficiently. By following the steps outlined, you can create a robust leave management system that dynamically updates leave balances based on user requests and approvals.