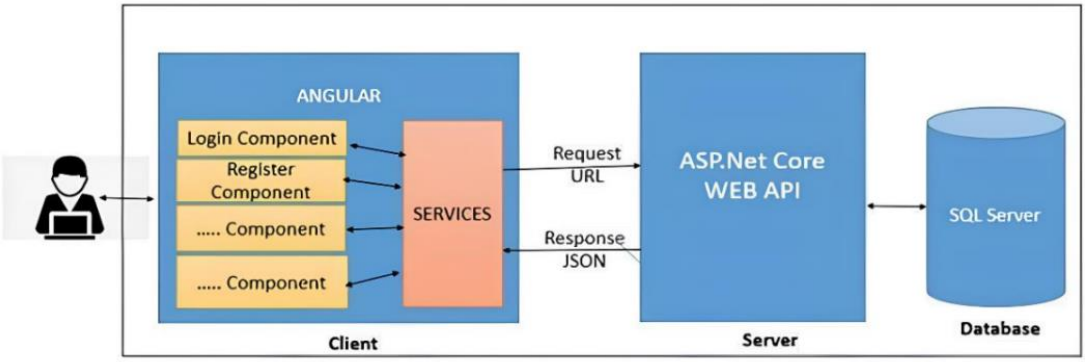
Introduction:

The WorkBuddy Application is a comprehensive software solution designed to streamline and enhance the interaction between employees and their managers within an organization. It aims to facilitate efficient communication, streamline administrative tasks, and provide valuable insights for improved decision-making. By centralizing employee-related processes and data, the application seeks to optimize productivity, foster collaboration, and promote transparency within the workplace. With user-friendly interfaces and robust backend functionality, the Employee-Manager Application empowers both employees and managers to effectively manage their responsibilities and achieve organizational goals.

Users of the System:

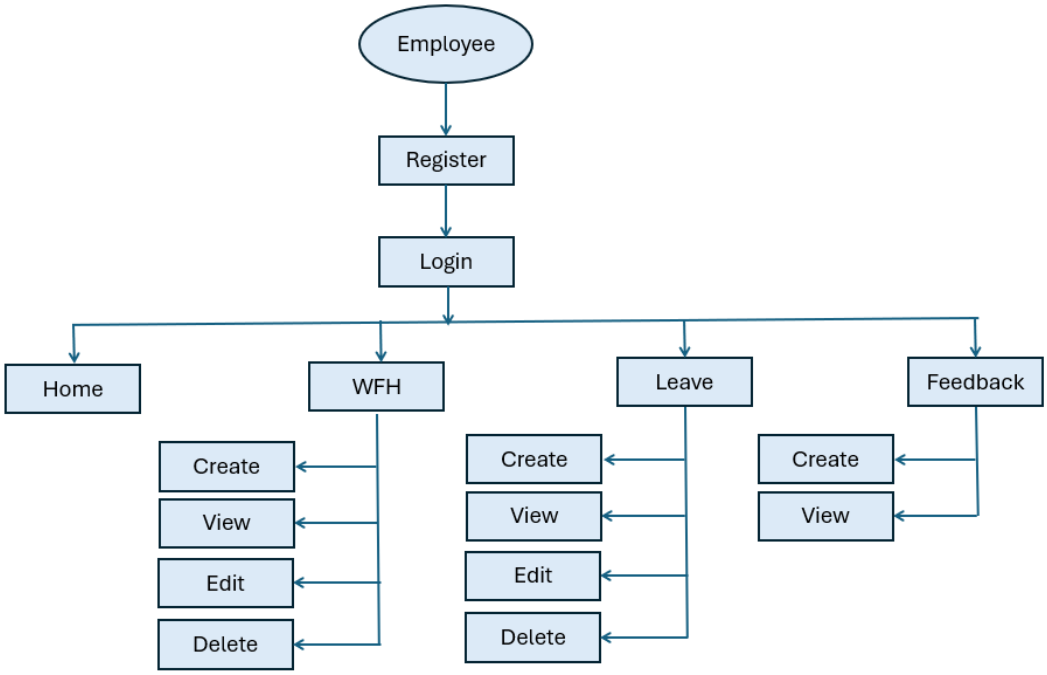
1. Manager
2. Employee

System Architecture Diagram:



Employee Actions

Flow diagram:



Leave Requests: Employees can request leaves for specific durations, providing the start and end dates of their absence along with a reason for the leave.

Leave History: Employees can view their past leave requests, including the status of each request (e.g., pending, approved, rejected).

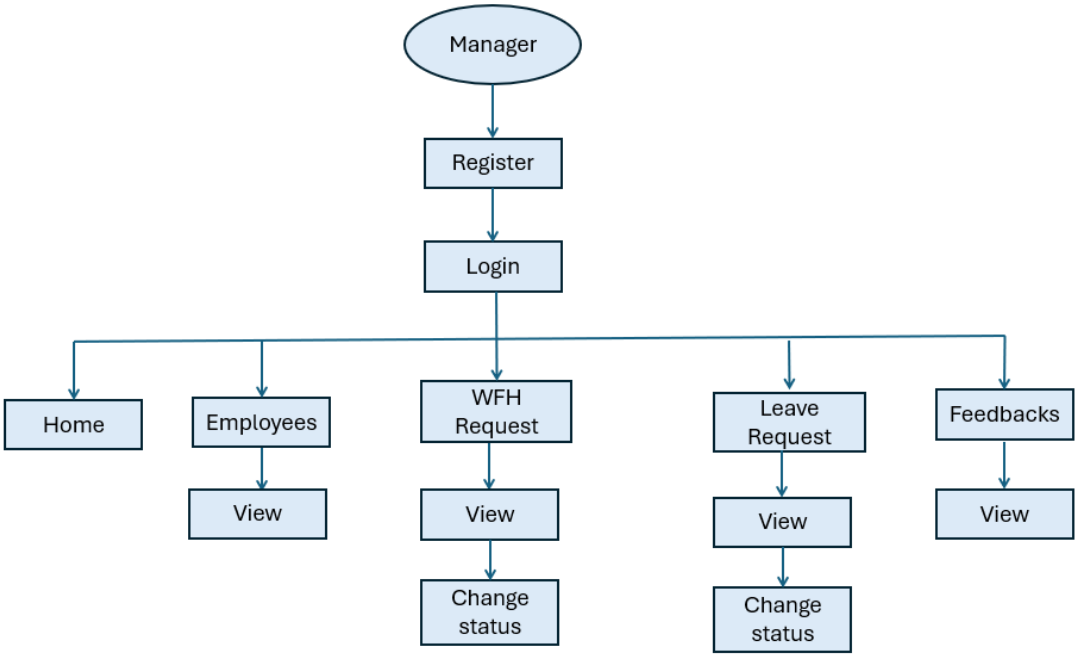
Work From Home (WFH) Requests: Employees can request to work from home for specified periods, providing the start and end dates of their remote work period along with a reason for the request.

WFH History: Employees can access their previous WFH requests and see the status of each request (e.g., pending, approved, rejected).

Feedback: Employees can view and post their feedback.

Manager Actions

Flow diagram:



Approval of Leave Requests: Managers can review leave requests submitted by employees. They have the authority to approve or reject leave requests based on company policies and workload considerations.

Approval of Work From Home (WFH) Requests: Managers can review and approve or reject WFH requests submitted by employees.

Employees: Managers can view all the employees.

Feedback: Managers can view all the employee feedback.

Modules of the Application:

Manager:

1. Register
2. Login
3. Home
4. Employees
5. WFH Request
6. Leave Request
7. Feedbacks

User:

1. Register
2. Login
3. Home
4. WFH
5. Leave
6. Feedback

Technology Stack

Front End

Angular 10+, HTML, CSS

Back End

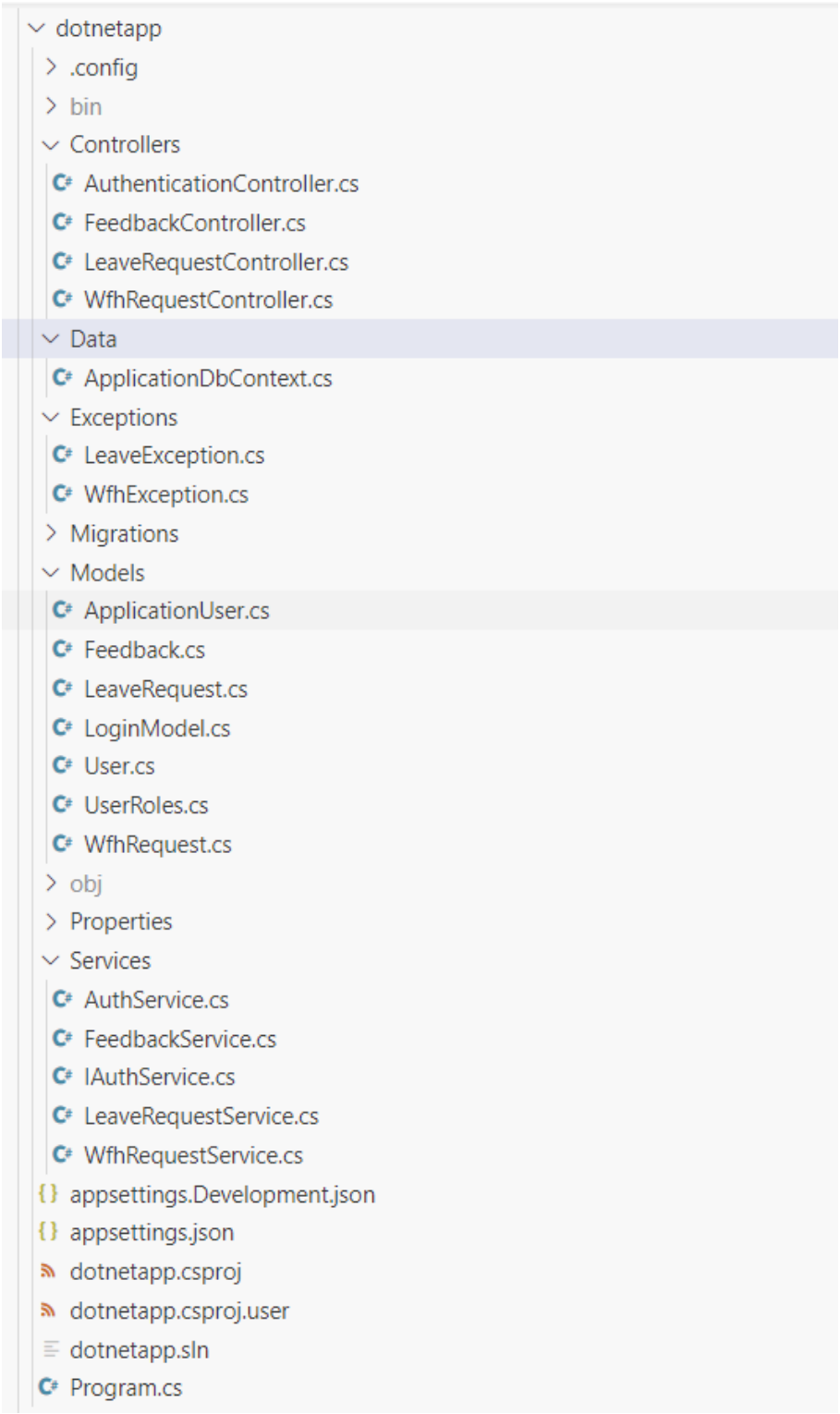
.NET Web API, EF Core, Microsoft SQL Server Database.

Application assumptions:

1. The login page should be the first page rendered when the application loads.
2. Manual routing should be restricted by implementing Auth Guard, utilizing the canActivate interface. For example, if the user enters as http://localhost:8080/dashboard or http://localhost:8080/user the page should not navigate to the corresponding page instead it should redirect to the login page.
3. Unless logged into the system, the user cannot navigate to any other pages.
4. Logging out must again redirect to the login page.

Backend Requirements:

Create folders named as **Models, Controllers, Services, Data and Exceptions** inside **dotnetapp** as mentioned in the below screenshot.



ApplicationDbContext: (**/Data/ApplicationDbContext.cs**)

Inside **Data** folder create **ApplicationDbContext** file with the following **DbSet** mentioned below.

public DbSet<User> Users { get; set; }

```
public DbSet<WfhRequest> WfhRequests { get; set; }
public DbSet<LeaveRequest> LeaveRequests { get; set; }
public DbSet<Feedback> Feedbacks { get; set; }
```

Model Classes:

Inside **Models** folder create all the model classes mentioned below.

Namespace: All the model classes should located within the **dotnetapp.Models** namespace.

User (Models / User.cs):
This class stores the user role (**Manager** or **Employee**) and all user information.
Properties:

- UserId: int
- Email: string
- Password: string
- Username: string
- MobileNumber: string
- UserRole: string (**Manager/Employee**)

Feedback (Models / Feedback.cs):
This class represents feedback submitted by users.
Properties:

- FeedbackId: int
- UserId: int
- User?: User
- FeedbackText: string
- Date: DateTime

LeaveRequest (Models / LeaveRequest.cs):

This class represents leave requests submitted by users.

Properties:

- LeaveRequestId: int
- UserId: int
- User?: User
- StartDate: DateTime
- EndDate: DateTime
- Reason: string

- LeaveType: string
- Status: string
- CreatedOn: DateTime

WfhRequest (Models / WfhRequest.cs):

This class represents work from home (WFH) requests submitted by users.

Properties:

- WfhRequestId: int
- UserId: int
- User?: User
- StartDate: DateTime
- EndDate: DateTime
- Reason: string
- Status: string
- CreatedOn: DateTime

LoginModel (Models / LoginModel.cs):

This class stores the email and password to authenticate the user during login.

Properties:

- Email: string
- Password: string

UserRoles (Models / UserRoles.cs):

This class defines constants for user roles.

Constants:

1. Manager: string
2. Employee: string

ApplicationUser (Models / ApplicationUser.cs):

This class represents a user in the application, inheriting from **IdentityUser** class.

Property:

- Name: string (Max length 30)

Exceptions:

WfhException (Exceptions / WfhException.cs)

1. Inside **Exceptions** folder create the exception file named **WfhException (WfhException.cs).**
2. **Purpose**: The **WfhException** class provides a mechanism for handling exceptions related to wfh operations within the application.

3. **Namespace**: It should located within the **dotnetapp.Exceptions** namespace.
4. **Inheritance**: Inherits from the base **Exception** class, enabling it to leverage existing exception handling mechanisms.
5. **Constructor**: Contains a constructor that accepts a message parameter, allowing to specify custom error messages when throwing exceptions.

Example Usage:

You might throw a **WfhException** in the following scenarios:

1. **When attempting to add a WFH request that overlaps with an existing one for the same user.**
2. **When trying to update a WFH request and the new dates overlap with another existing WFH request for the same user.**

LeaveException (Exceptions / LeaveException.cs)

6. Inside **Exceptions** folder create the exception file named **LeaveException (LeaveException.cs).**
7. **Purpose**: The **LeaveException** class provides a mechanism for handling exceptions related to leave operations within the application.
8. **Namespace**: It should located within the **dotnetapp.Exceptions** namespace.
9. **Inheritance**: Inherits from the base **Exception** class, enabling it to leverage existing exception handling mechanisms.
10. **Constructor**: Contains a constructor that accepts a message parameter, allowing to specify custom error messages when throwing exceptions.

Example Usage:

You might throw a **LeaveException** in the following scenarios:

1. **When attempting to add a leave request that overlaps with an existing one for the same user.**
2. **When trying to update a leave request and the new dates overlap with another existing leave request for the same user.**

Important note:
Implement database logic only in the **service file functions without using try-catch**. Use **try-catch only in the controller files** and call the service file functions inside it.

Services:
Inside "**Services"** folder create all the services file mentioned below.

Namespace: All the services file should located within the **dotnetapp.Services** namespace.

WfhRequestService (Services / WfhRequestService.cs):

This service class provides methods to interact with feedback data stored in the database.

Constructor:

```
public WfhRequestService(ApplicationDbContext context)
    {
        _context = context;
    }
```

Functions:

1. **public async Task<IEnumerable<WfhRequest>> GetAllWfhRequests():**
    - Retrieves and returns all work-from-home (WFH) requests from the database.
2. **public async Task<WfhRequest> GetWfhRequestById(int wfhRequestId):**
    - Retrieves a work-from-home (WFH) request from the database with the specified wfhRequestId.
3. **public async Task<bool> AddWfhRequest(WfhRequest wfhRequest):**
    - Check if a work-from-home (WFH) request with overlapping dates already exists in the database for the same user.
    - If a WFH request with overlapping dates exists, throw a **WfhException** with the message "**There is already a WFH request for these dates**".
    - If no overlapping WFH request exists, add the new WFH request to the database.
    - Save changes asynchronously to the database.
    - Returns true for the successful insertion.
4. **public async Task<bool> UpdateWfhRequest(int wfhRequestId, WfhRequest wfhRequest):**
    - If no work-from-home (WFH) request with the specified wfhRequestId is found in the database, return false.

- Check if a work-from-home (WFH) request with overlapping dates already exists in the database for the same user.
- If a WFH request with overlapping dates exists, throw a **WfhException** with the message "**There is already a WFH request for these dates**".
- If no overlapping WFH request exists, update the existing WFH request with the values from the provided **wfhRequest** object.
- Save changes asynchronously to the database.
- Return true for the successful update.

5. **public async Task<bool> DeleteWfhRequest(int wfhRequestId):**
    - Retrieve the work-from-home (WFH) request from the database based on the provided **wfhRequestId**.
    - If no wfhRequest with the specified **wfhRequestId** is found, return false.
    - If found, delete the wfhRequest with the provided **wfhRequestId**.
    - Save changes asynchronously to the database.
    - Return true for the successful deletion.
6. **public async Task<IEnumerable<WfhRequest>> GetWfhRequestsByUserId(int userId):**
    - Retrieves and returns all work-from-home (WFH) requests from the database for the specified **userId**.

LeaveRequestService (Services / LeaveRequestService.cs):

This service class provides methods to interact with feedback data stored in the database.

Constructor:

```
public LeaveRequestService(ApplicationDbContext context)
{
    _context = context;
}
```

Functions:

1. **public async Task<IEnumerable<LeaveRequest>> GetAllLeaveRequests():**
    - Retrieves and returns all leave requests from the database.

2. **public async Task<LeaveRequest> GetLeaveRequestById(int leaveRequestId):**
    - Retrieves a leave request from the database with the specified leaveRequestId.
3. **public async Task<bool> AddLeaveRequest(LeaveRequest leaveRequest):**
    - Check if a leave request with overlapping dates already exists in the database for the same user.
    - If a leave request with overlapping dates exists, throw a **LeaveException** with the message "**There is already a leave request for these dates**".
    - If no overlapping leave request exists, add the new leave request to the database.
    - Save changes asynchronously to the database.
    - Returns true for the successful insertion.
4. **public async Task<bool> UpdateLeaveRequest(int leaveRequestId, LeaveRequest leaveRequest):**
    - If no leave request with the specified **leaveRequestId** is found in the database, return false.
    - Check if a leave request with overlapping dates already exists in the database for the same user.
    - If a leave request with overlapping dates exists, throw a **LeaveException** with the message "**There is already a leave request for these dates**".
    - If no overlapping leave request exists, update the existing leave request with the values from the provided leaveRequest object.
    - Save changes asynchronously to the database.
    - Return true for the successful update.
5. **public async Task<bool> DeleteLeaveRequest(int leaveRequestId):**
    - Retrieve the leave request from the database based on the provided leaveRequestId.
    - If no **leaveRequest** with the specified **leaveRequestId** is found, return false.
    - If found, delete the **leaveRequest** with the provided **leaveRequestId**.
    - Save changes asynchronously to the database.
    - Return true for the successful deletion.

6. **public async Task<IEnumerable<LeaveRequest>> GetLeaveRequestsByUserId(int userId):**
    - Retrieves and returns all leave requests from the database for the specified userId.

FeedbackService (Services / FeedbackService.cs):
This service class provides methods to interact with feedback data stored in the database.

Constructor:
public **FeedbackService**(ApplicationDbContext context)
{
_context = context;
}

Functions:
1. **public async Task<IEnumerable<Feedback>> GetAllFeedbacks():**
    - Retrieves all feedbacks from the database.

2. **public async Task<IEnumerable<Feedback>> GetFeedbacksByUserId(int userId):**
    - Retrieves all feedbacks associated with a specific **userId** from the database.

3. **public async Task<bool> AddFeedback(Feedback feedback):**
    - Adds new feedback to the database.
    - Return **true** for the successful insertion.

4. **public async Task<bool> DeleteFeedback(int feedbackId):**
    - Retrieve the existing feedback from the database with the specified **feedbackId**.
    - If no feedback with the specified feedbackId is found, return **false**.
    - If found, delete the feedback with the provided feedbackId.
    - Save changes asynchronously to the database.
    - Return **true** for the successful delete.

AuthService (Services / AuthService.cs):

The **AuthService** class is responsible for user authentication and authorization.

Constructor:

public **AuthService**(UserManager<ApplicationUser> userManager,
RoleManager<IdentityRole> roleManager, IConfiguration configuration,
ApplicationDbContext context)

{

this.userManager = userManager;

this.roleManager = roleManager;

_configuration = configuration;

_context = context;

}

Functions:

1. public async Task<(int, string)> Registration (User model, string role):

- Check if the email already exists in the database. If so return "**User already exists**".
- Registers a new user with the provided details and assigns a role.
- If any error occurs return "**User creation failed! Please check user details and try again**".
- Return "**User created successfully!"** for the successful register.

2. public async Task<(int, string)> Login (LoginModel model):

- Find user by email in the database.
- Check if user exists, if not return "**Invalid email**".
- If the user exists, check the password is correct, if not return "**Invalid password**".
- Logs in a user with the provided credentials and generates a **JWT** token for authentication.

3. private string GenerateToken(IEnumerable<Claim> claims):

- Generates a JWT token based on the provided claims.

4. **public async Task<List<User>> GetAllEmployees()**

- Retrieves and returns all users who have the role of "**Employee**" from the database.

IAuthService (Services / IAuthService.cs):

The **IAuthService** is an interface that defines methods for user registration and login.

Methods:

1. Task< (int, string)> Registration (User model, string role);
2. Task< (int, string)> Login (LoginModel model);
3. Task<List<User>> GetAllEmployees();

<u>Controllers</u>:

Inside "**Controllers"** folder create all the controllers file mentioned below.

Namespace: All the controllers file should located within the **dotnetapp.Controllers** namespace.

AuthenticationController (Controllers / AuthenticationController.cs):

This controller handles user authentication and registration requests.

Functions:

1. public async Task<IActionResult> Login(LoginModel model)

a. Accepts login requests, validates the payload, and calls the authentication service to perform user login.

b. It utilizes **_authService.Login(model)** method.

c. Returns a **200 OK response** with a JWT token upon successful login.

d. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

2. public async Task<IActionResult> Register(User model):

a. Accepts registration requests, validates the payload. If fails, then returns error.

b. Calls the authentication service to register a new user(**_authService.Registration(model, model.UserRole)**). Returns a **200 OK response with** success message upon successful registration.

c. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

3. public async Task<IActionResult> GetAllEmployees()

a. Retrieves a list of all employees from the database.

b. Utilizes the **_authService.GetAllEmployees()** method to fetch the employees.

c. Returns a 200 OK response along with the list of employees upon successful retrieval.

d. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

LeaveRequestController (Controllers / LeaveRequestController.cs):

This controller manages leave requests, interacting with the **LeaveRequestService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<LeaveRequest>>> GetAllLeaveRequests():

a. Implements the logic inside a try-catch block.

b. The **GetAllLeaveRequests** method is a controller action responsible for retrieving all leave requests.

c. It calls the **_leaveRequestService.GetAllLeaveRequests()** method to fetch all leave requests from the service layer.

d. It returns a 200 OK response with the retrieved leave requests upon successful retrieval.

2. public async Task<ActionResult<LeaveRequest>> GetLeaveRequestById(int leaveRequestId):

a. Implements the logic inside a try-catch block.

b. The GetLeaveRequestById method is a controller action responsible for retrieving a leave request by its ID.

c. It calls the **_leaveRequestService.GetLeaveRequestById(leaveRequestId)** method to retrieve the leave request from the service layer.

d. If the leave request is not found, it returns a 404 Not Found response with a message “Cannot find any leave request”.

e. If the leave request is found, it returns a 200 OK response with the leave request data.

3. public async Task<ActionResult> AddLeaveRequest([FromBody] LeaveRequest leaveRequest):

a. Implements the logic inside a try-catch block.

b. The AddLeaveRequest method is a controller action responsible for adding a leave request.

c. It receives the leave request data in the request body.

d. It tries to add the leave request using the **_leaveRequestService.AddLeaveRequest(leaveRequest)** method.

e. If adding the leave request is successful, it returns a 200 OK response with a success message “Leave request added successfully”.

f. If adding the leave request fails, it returns a 500 Internal Server Error response with a failure message “Failed to add leave request”.

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdateLeaveRequest(int leaveRequestId, [FromBody] LeaveRequest leaveRequest):

a. Implements the logic inside a try-catch block.

b. The UpdateLeaveRequest method is a controller action responsible for updating a leave request.

c. It receives the leave request ID and updated leave request data in the request body.

d. It tries to update the leave request using the **_leaveRequestService.UpdateLeaveRequest(leaveRequestId, leaveRequest)** method.

e. If the update is successful, it returns a 200 OK response with a success message “Leave request updated successfully”.

f. If the leave request is not found, it returns a 404 Not Found response with a message “Cannot find any leave request”.

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> DeleteLeaveRequest(int leaveRequestId):

a. Implements the logic inside a try-catch block.

b. The **DeleteLeaveRequest** method is a controller action responsible for deleting a leave request.

c. It receives the leave request ID to be deleted.

d. It tries to delete the leave request using the **_leaveRequestService.DeleteLeaveRequest(leaveRequestId)** method.

e. If the deletion is successful, it returns a 200 OK response with a success message "**Leave request deleted successfully**".

f. If the leave request is not found, it returns a 404 Not Found response with a message "**Cannot find any leave request**".

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

WfhRequestController (Controllers / WfhRequestController.cs):

This controller manages work from home (WFH) requests, interacting with the **WfhRequestService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<WfhRequest>>> GetAllWfhRequests():

a. Implements the logic inside a try-catch block.

b. The GetAllWfhRequests method is a controller action responsible for retrieving all WFH requests.

c. It calls the **_wfhRequestService.GetAllWfhRequests()** method to fetch all WFH requests from the service layer.

d. It returns a 200 OK response with the retrieved WFH requests upon successful retrieval.

2. public async Task<ActionResult<WfhRequest>> GetWfhRequestById(int wfhRequestId):

a. Implements the logic inside a try-catch block.

b. The GetWfhRequestById method is a controller action responsible for retrieving a WFH request by its ID.

c. It calls the **_wfhRequestService.GetWfhRequestById(wfhRequestId)** method to retrieve the WFH request from the service layer.

d. If the WFH request is not found, it returns a 404 Not Found response with a message "Cannot find any WFH request".

e. If the WFH request is found, it returns a 200 OK response with the WFH request data.

3. public async Task<ActionResult> AddWfhRequest([FromBody] WfhRequest wfhRequest):

a. Implements the logic inside a try-catch block.

b. The AddWfhRequest method is a controller action responsible for adding a WFH request.

c. It receives the WFH request data in the request body.

d. It tries to add the WFH request using the **_wfhRequestService.AddWfhRequest(wfhRequest)** method.

e. If adding the WFH request is successful, it returns a 200 OK response with a success message “WFH request added successfully”.

f. If adding the WFH request fails, it returns a 500 Internal Server Error response with a failure message “Failed to add WFH request”.

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdateWfhRequest(int wfhRequestId, [FromBody] WfhRequest wfhRequest):

a. Implements the logic inside a try-catch block.

b. The UpdateWfhRequest method is a controller action responsible for updating a WFH request.

c. It receives the WFH request ID and updated WFH request data in the request body.

d. It tries to update the WFH request using the **_wfhRequestService.UpdateWfhRequest(wfhRequestId, wfhRequest)** method.

e. If the update is successful, it returns a 200 OK response with a success message “WFH request updated successfully”.

f. If the WFH request is not found, it returns a 404 Not Found response with a message “Cannot find any WFH request”.

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> DeleteWfhRequest(int wfhRequestId):

a. Implements the logic inside a try-catch block.

b. The DeleteWfhRequest method is a controller action responsible for deleting a WFH request.

c. It receives the WFH request ID to be deleted.

d. It tries to delete the WFH request using the **_wfhRequestService.DeleteWfhRequest(wfhRequestId)** method.

e. If the deletion is successful, it returns a 200 OK response with a success message “WFH request deleted successfully”.

f. If the WFH request is not found, it returns a 404 Not Found response with a message “Cannot find any WFH request”.

g. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

FeedbackController (Controllers / FeedbackController.cs):

This controller manages feedbacks, interacting with the **FeedbackService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<Feedback>>> GetAllFeedbacks():

a. Implement the logic inside **try-catch block.**

b. The **GetAllFeedbacks** method is a controller action responsible for retrieving all feedbacks.

c. It tries to get all feedbacks using the **_feedbackService.GetAllFeedbacks()** method.

d. If the operation is successful, it returns a **200 OK response** with the retrieved feedbacks.

e. If an exception occurs during the process, it returns a **500 Internal Server Error response** with the exception message.

2. public async Task<ActionResult<IEnumerable<Feedback>>> GetFeedbacksByUserId(int userId):

a. Implement the logic inside **try-catch block**.

b. The **GetFeedbacksByUserId** method is a controller action responsible for retrieving feedbacks by **userId**.

c. It tries to get feedbacks by userId using the **_feedbackService.GetFeedbacksByUserId(userId)** method.

d. If feedbacks are found, it returns a **200 OK response** with the retrieved feedbacks.

e. If an exception occurs during the process, it returns a **500 Internal Server Error response** with the exception message.

3. public async Task<ActionResult> AddFeedback([FromBody] Feedback feedback):

a. Implement the logic inside **try-catch block**.

b. The **AddFeedback** method is a controller action responsible for adding a new feedback.

c. It receives the feedback data in the request body.

d. It tries to add the feedback using the **_feedbackService.AddFeedback(feedback)** method.

e. If adding the feedback is successful, it returns a **200 OK response** with a success message "**Feedback added successfully**".

f. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

4. public async Task<ActionResult> DeleteFeedback(int feedbackId):

a. Implement the logic inside **try-catch block**.

b. The **DeleteFeedback** method is a controller action responsible for deleting a feedback.

c. It receives the **feedbackId** to be deleted.

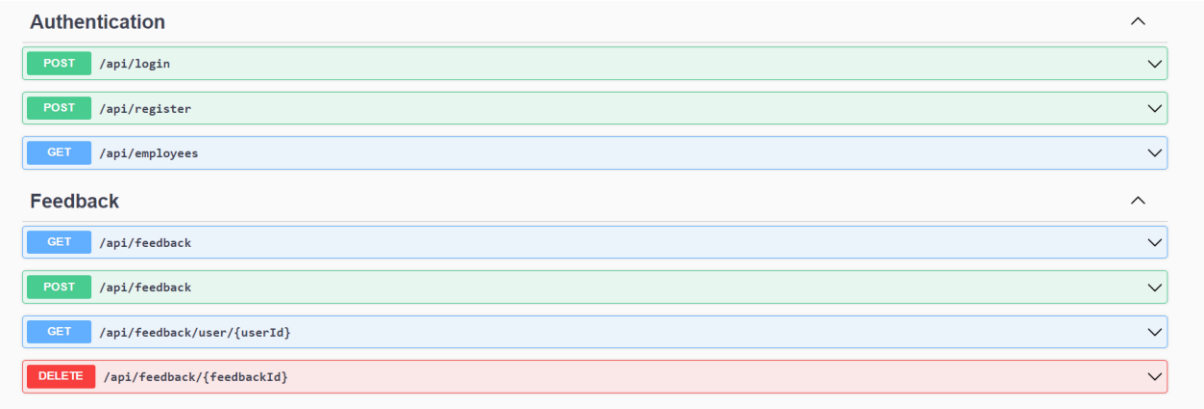
d. It tries to delete the feedback using the **_feedbackService.DeleteFeedback(feedbackId)** method.

e. If the deletion is successful, it returns a **200 OK response** with a success message "**Feedback deleted successfully**".

f. If the feedback is not found, it returns a **404 Not Found** response with a message "**Cannot find any feedback**".

g. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

Endpoints:



Note:

Ensure that JWT authentication is enabled for all endpoints, with authorization based on user roles. For example, the endpoint for adding a leave/wfh should only be accessible with a JWT present in the header. If not, return **Unauthorized**. Additionally, it should be restricted to users with the "**Employee"** role. If a user **without the "Employee" role (Manager)** attempts to access it, it should return a **Forbidden** error.

1. **Login:** [Access for both Employee and Manager]
   Endpoint name: "/api/login"
   Method: POST
   Request body: {
   "Email": "string",
   "Password": "string"
   }

Response:

Status Code	Response body

200 (HttpStatusCode OK)	JSON object containing a JWT token.
	Example: {
	"Status": "Success",
	"token":
	"eyJhbGciOiJIUzI1NiiQWRtaW4iLCJqdGkiOiJmZTUyYzAxZS1"
	}

400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

2. Register: [Access for both Employee and Manager]

Endpoint name: "/api/register"

Method: POST

Request body:

```
{
  "Username": "string",
  "Email": "user@example.com",
  "MobileNumber": "9876541221",
  "Password": "Pass@2425",
  "UserRole": "string"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

3. Get all employees: [Access for only **Manager**]

Endpoint name: "/api/employees"

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all the employees.

500	JSON object containing Error message.

4. <u>Get all leave request:</u> [Access for only **Manager**]

Endpoint name: “/api/leaverequest”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all the leaverequest.
500	JSON object containing Error message.

5. <u>Add leave request:</u> [Access for only Employee]

Endpoint name: “/api/leaverequest”

Method: POST

Request body:


```
{
  "UserId": 1,
  "StartDate": "2024-05-19T01:58:41.097Z",
  "EndDate": "2024-05-19T01:58:41.097Z",
  "Reason": "string",
  "LeaveType": "string",
  "Status": "string",
  "CreatedOn": "2024-05-19T01:58:41.097Z"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

6. Get specific leave request: [Access for only Employee]

Endpoint name: “/api/leaverequest /{ leaveRequestId }”

Method: GET

Parameter: leaveRequestId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing leave request details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

7. Update leave request: [Access for both Employee and Manager]

Endpoint name: “/api/leaverequest /{leaveRequestId}”

Method: PUT

Parameter: leaveRequestId

Request body:

```
{
  "UserId": 1,
  "StartDate": "2024-05-19T01:58:41.097Z",
  "EndDate": "2024-05-19T01:58:41.097Z",
  "Reason": "string",
  "LeaveType": "string",
  "Status": "string",
  "CreatedOn": "2024-05-19T01:58:41.097Z"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing leave request details.

404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

8. Delete leave request: [Access for only Employee]

Endpoint name: "/api/leaverequest/{leaveRequestId}"

Method: DELETE

Parameter: leaveRequestId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

9. Get leave requests specific to user: [Access for only Employee]

Endpoint name: "/api/leaverequest /user/{userId}"

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing leave request details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

10. Get all wfh request: [Access for only **Manager**]

Endpoint name: "/api/wfhrequest"

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all the wfhrequest.

500	JSON object containing Error message.

11. Add wfh request: [Access for only Employee]

Endpoint name: "/api/wfhrequest"

Method: POST

Request body:

```
{
  "UserId": 1,
  "StartDate": "2024-05-19T01:58:41.097Z",
  "EndDate": "2024-05-19T01:58:41.097Z",
  "Reason": "string",
  "Status": "string",
  "CreatedOn": "2024-05-19T01:58:41.097Z"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

12. Get specific wfh request: [Access for only Employee]

Endpoint name: "/api/wfhrequest /{wfhRequestId}"

Method: GET

Parameter: wfhRequestId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing wfh request details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

13. <u>Update wfh request:</u> [Access for both Employee and Manager]

Endpoint name: “/api/wfhrequest /{wfhRequestId}”

Method: PUT

Parameter: wfhRequestId

Request body:


```
{
  "UserId": 1,
  "StartDate": "2024-05-19T01:58:41.097Z",
  "EndDate": "2024-05-19T01:58:41.097Z",
  "Reason": "string",
  "Status": "string",
  "CreatedOn": "2024-05-19T01:58:41.097Z"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing wfh request details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

14. <u>Delete wfh request:</u> [Access for only Employee]

Endpoint name: “/api/wfhequest/{ wfhRequestId }”

Method: DELETE

Parameter: wfhRequestId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

15. Get wfh requests specific to user: [Access for only Employee]

Endpoint name: "/api/wfhrequest /user/{userId}"

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing wfh request details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

16. Get all feedbacks: [Access for only Manager]

Endpoint name: "/api/feedback"

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all feedback.
500	JSON object containing Error message.

17. Add feedback: [Access for only Employee]

Endpoint name: "/api/feedback"

Method: POST

Request body:

```
{
  "UserId": 0,
  "FeedbackText": "string",
  "Date": "2024-07-07T12:28:56.927Z"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

18. <u>Get feedback specific to user</u>: [Access for only Employee]

Endpoint name: "/api/feedback/{userId }"

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing feedback details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

19. <u>Delete feedback</u>: [Access for only Employee]

Endpoint name: "/api/feedback /{feedbackId}"

Method: DELETE

Parameter: feedbackId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

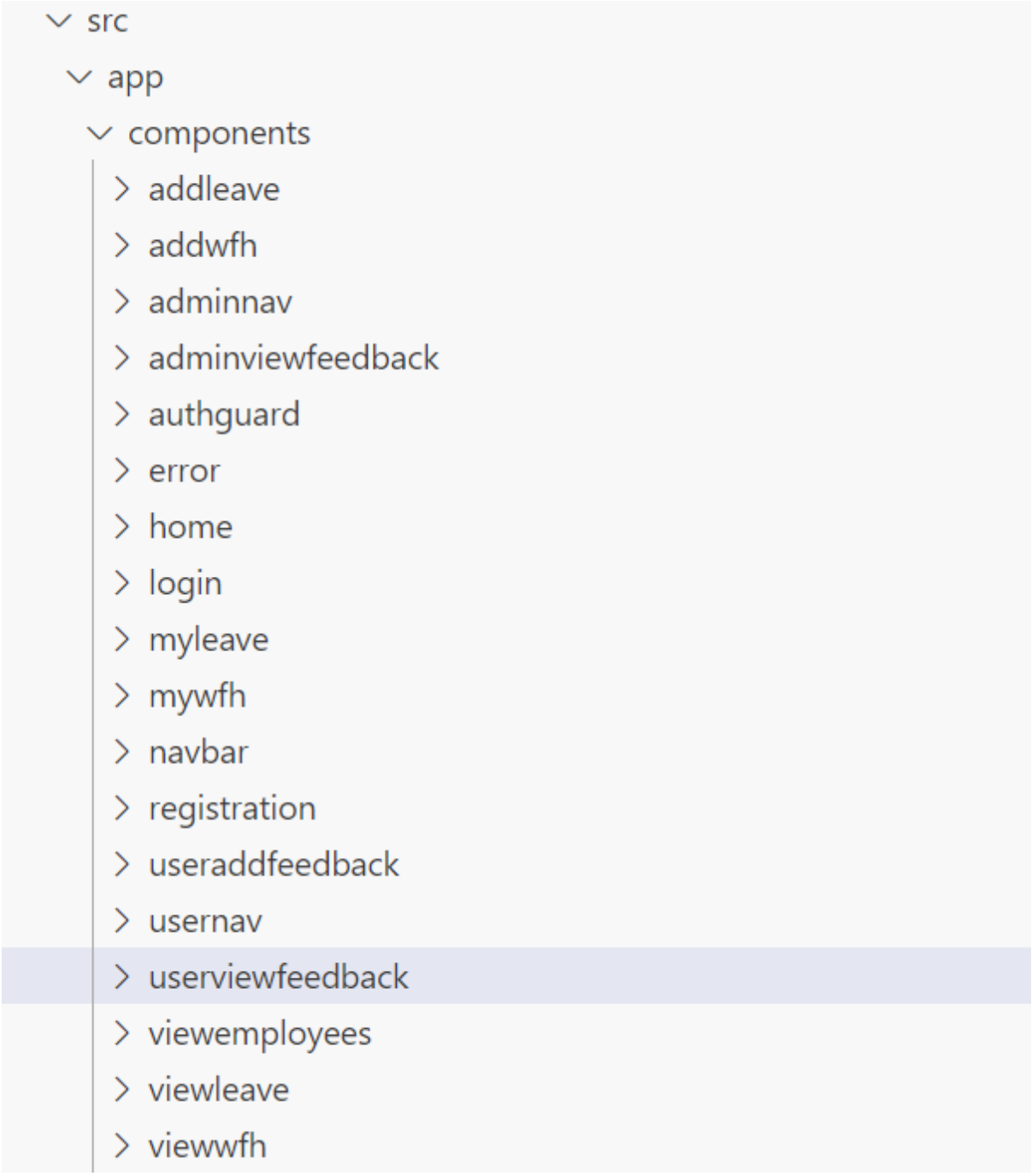
<u>Frontend Requirements:</u>

- Create a folder named components inside the app to store all the components. (Refer project structure screenshots).

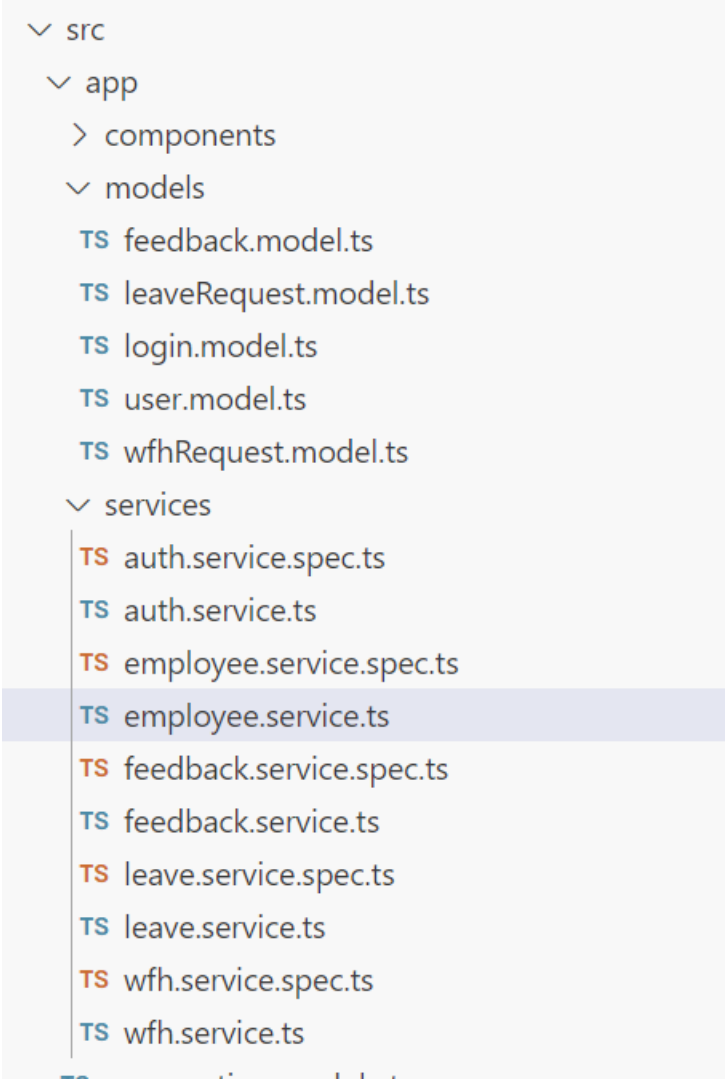
- Create a folder named models inside app to store all the model interface.
- Create a folder named as services inside app to implement all the services.
- Create model interface referring the backend entities mentioned in the backend requirements accordingly.
- You can create your own components based on the application requirements.
- Import model files, services and components as required.

Project Folder Screenshot:

Components:



Services and Models:



Frontend Models:

User Model:

class User {

UserId?: number;

Email: string;

Password: string;

Username: string;

MobileNumber: string;

UserRole: string;

}

Login Model:

class Login {

Email: string;

Password: string;

}

LeaveRequest Model:

class LeaveRequest {

LeaveRequestId?: number;

UserId: number;

StartDate: Date;

EndDate: Date;

Reason: string;

LeaveType: string;

Status: string;

CreatedOn: Date;

}

WfhRequest Model:

class WfhRequest {

WfhRequestId?: number;

UserId: number;

StartDate: Date;

EndDate: Date;

Reason: string;

Status: string;

CreatedOn: Date;

}

Feedback Model:

class Feedback {

FeedbackId?: number;

UserId: number;

FeedbackText: string;

Date: Date;

}

Frontend services:

- Declare a public property apiUrl to store the backend URL in all the services.
- For example, public apiUrl = 'http://localhost:8080'. Instead of 'localhost', replace it with the URL of your workspace port 8080 URL.
- For the API's to be used please refer the API Table.
- Authorized token to be passed in headers for all end points.
- **Important note:** Authorization token should be prefixed with 'Bearer'.

WfhService (services / wfh.service.ts):

Create a service named WfhService inside the app/services folder to implement the following functions.

Methods Overview:

1. **getAllWfhRequests(): Observable<WfhRequest[]>:**
    a. Use this method to retrieve all work from home (WFH) requests. It sends a GET request to the '/api/wfhrequest' endpoint with the JWT token in the authorization header.

2. **getWfhRequestById(wfhRequestId: number): Observable<WfhRequest>:**
    a. This method is used to retrieve a specific WFH request by its ID. It sends a GET request to the '/api/wfhrequest/{wfhRequestId}' endpoint with the JWT token in the authorization header.
3. **addWfhRequest(requestObject: WfhRequest): Observable<WfhRequest>:**
    a. Use this method to add a new WFH request. It sends a POST request to the '/api/wfhrequest' endpoint with the WFH request object provided as the body and the JWT token in the authorization header.
4. **updateWfhRequest(wfhRequestId: number, requestObject: WfhRequest): Observable<WfhRequest>:**
    a. This method is used to update an existing WFH request. It sends a PUT request to the '/api/wfhrequest/{wfhRequestId}' endpoint with the updated WFH request object provided as the body and the JWT token in the authorization header.
5. **deleteWfhRequest(wfhRequestId: number): Observable<void>:**
    a. Use this method to delete a specific WFH request by its ID. It sends a DELETE request to the '/api/wfhrequest/{wfhRequestId}' endpoint with the JWT token in the authorization header.
6. **getWfhRequestsByUserId(userId: number): Observable<WfhRequest[]>:**
    a. This method is used to retrieve all WFH requests associated with a specific user ID. It sends a GET request to the '/api/wfhrequest/user/{userId}' endpoint with the JWT token in the authorization header.

LeaveService (services / leave.service.ts):

Create a service named LeaveService inside the app/services folder to implement the following functions.

Methods Overview:

1. **getAllLeaveRequests(): Observable<LeaveRequest[]>:**
    a. Use this method to retrieve all leave requests. It sends a GET request to the '/api/leaverequest' endpoint with the JWT token in the authorization header.

2. **getLeaveRequestById(leaveRequestId: number): Observable<LeaveRequest>:**
    a. This method is used to retrieve a specific leave request by its ID. It sends a GET request to the '/api/leaverequest/{leaveRequestId}' endpoint with the JWT token in the authorization header.

3. **addLeaveRequest(requestObject: LeaveRequest): Observable<LeaveRequest>:**
    a. Use this method to add a new leave request. It sends a POST request to the '/api/leaverequest' endpoint with the leave request object provided as the body and the JWT token in the authorization header.

4. **getLeaveRequestsByUserId(userId: number): Observable<LeaveRequest[]>:**
    a. This method is used to retrieve all leave requests associated with a specific user ID. It sends a GET request to the '/api/leaverequest/user/{userId}' endpoint with the JWT token in the authorization header.

5. **updateLeaveRequest(leaveRequestId: number, requestObject: LeaveRequest): Observable<LeaveRequest>:**
    a. Use this method to update an existing leave request. It sends a PUT request to the '/api/leaverequest/{leaveRequestId}' endpoint with the updated leave request object provided as the body and the JWT token in the authorization header.

6. **deleteLeaveRequest(leaveRequestId: number): Observable<void>:**
    a. This method is used to delete a specific leave request by its ID. It sends a DELETE request to the '/api/leaverequest/{leaveRequestId}' endpoint with the JWT token in the authorization header.

EmployeeService (services / employee.service.ts):

Create a service named **EmployeeService** inside the **app/services** folder to implement the following function.

Methods Overview:

1. **getAllEmployees(): Observable<User[]>:** Use this method to retrieve all employees. It sends a GET request to the '/api/employees' endpoint with the JWT token in the authorization header.

AuthService(services/auth.service.ts):

Create a service name as **auth** inside app/services folder to implement the following functions.

Methods Overview:

1. **register(user: User): Observable<any>:**
    a. Use this method to register a new user. It sends a POST request to the '/api/register' endpoint with the user data provided as the body.
2. **login(login : Login): Observable<any>:**
    a. This method is used to authenticate a user by logging them in. It sends a POST request to the '/api/login' endpoint with the user's email and password. Upon successful login, it stores the JWT token in localStorage and updates the user's role and ID using BehaviorSubjects.

FeedbackService(feedback.service.ts):

Create a service name as feedback inside app/services and implement the following functions in it.

Methods Overview:

1. **sendFeedback(feedback: Feedback): Observable<Feedback>:**
    a. Use this method to send feedback to the server. It sends a POST request to the '/api/feedback' endpoint with the feedback data provided and the authorization token prefixed with 'Bearer' stored in localStorage.
2. **getAllFeedbacksByUserId(userId: string): Observable<Feedback[]>:**
    a. This method retrieves all feedbacks submitted by a specific user. It sends a GET request to the '/api/feedback/user/:userId' endpoint with the user ID and the authorization token prefixed with 'Bearer' stored in localStorage.

3. **deleteFeedback(feedbackId: string): Observable<void>:**
    a. Call this method to delete a feedback with the specified ID. It sends a DELETE request to the '/api/feedback/:feedbackId' endpoint with the feedback ID and the authorization token prefixed with 'Bearer' stored in localStorage.
4. **getFeedbacks(): Observable<Feedback[]>:**
    a. This method fetches all feedbacks from the server. It sends a GET request to the '/api/feedback' endpoint with the authorization token prefixed with 'Bearer' stored in localStorage.

Validations:

Client-Side Validation:

- Implement client-side validation using HTML5 attributes and JavaScript to validate user input before making API requests.
- Provide immediate feedback to users for invalid input, such as displaying error messages near the input fields.

Server-Side Validation:

- Implement server-side validation in the controllers to ensure data integrity.
- Validate user input and API responses to prevent unexpected or malicious data from affecting the application.
- Return appropriate validation error messages to the user interface for any validation failures.

Exception Handling:

- Implement exception handling mechanisms in the controllers to gracefully handle errors and exceptions. Define custom exception classes for different error scenarios, such as API communication errors or database errors.
- Log exceptions for debugging purposes while presenting user-friendly error messages to users. Record all the exceptions and errors handled store in separate table “ErrorLogs”.

Error Pages:

Create custom error pages for different HTTP status codes (e.g., 404 Not Found, 500 Internal Server Error) to provide a consistent and user-friendly error experience. Ensure that error pages contain helpful information and guidance for users.

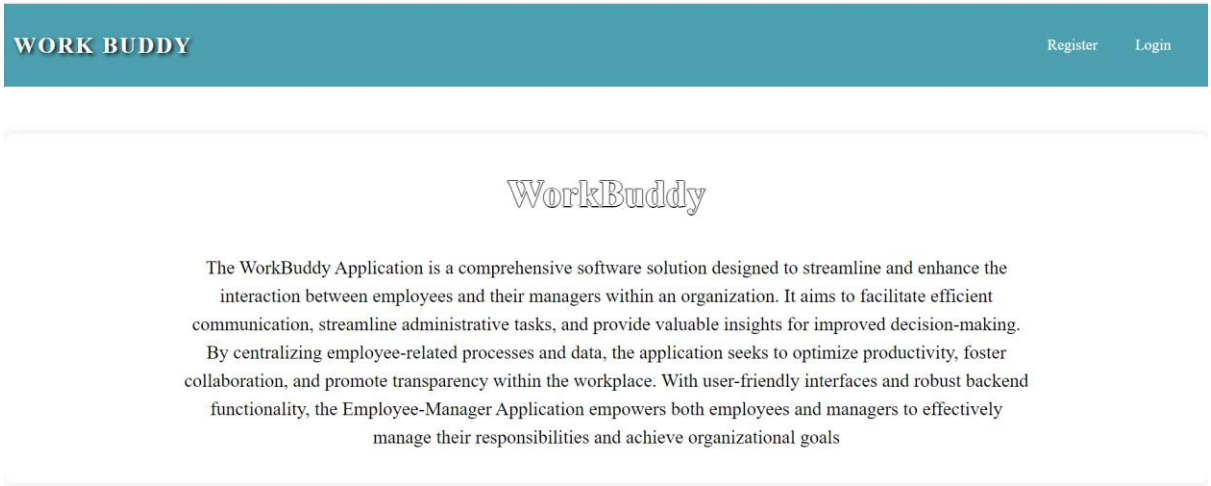
Thus, create a reliable and user-friendly web application that not only meets user expectations but also provides a robust and secure experience, even when faced with unexpected situations. Error page has to be displayed if something goes wrong.

Frontend Screenshots:
• All the asterisk (*) marked fields are mandatory in the form. Make sure to mark all the field names with * symbol followed by the validations.

Navigation Bar:
(**navbar component**)

Component displays a title along with router links to "Register" and "Login".

Landing Page:
(**home component**)

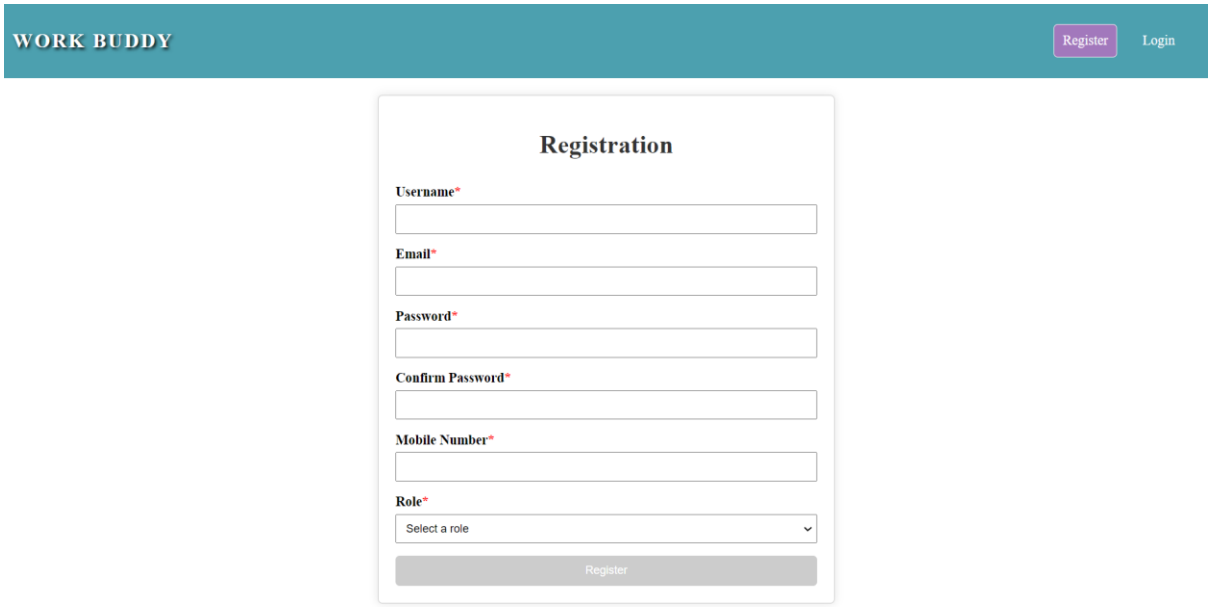


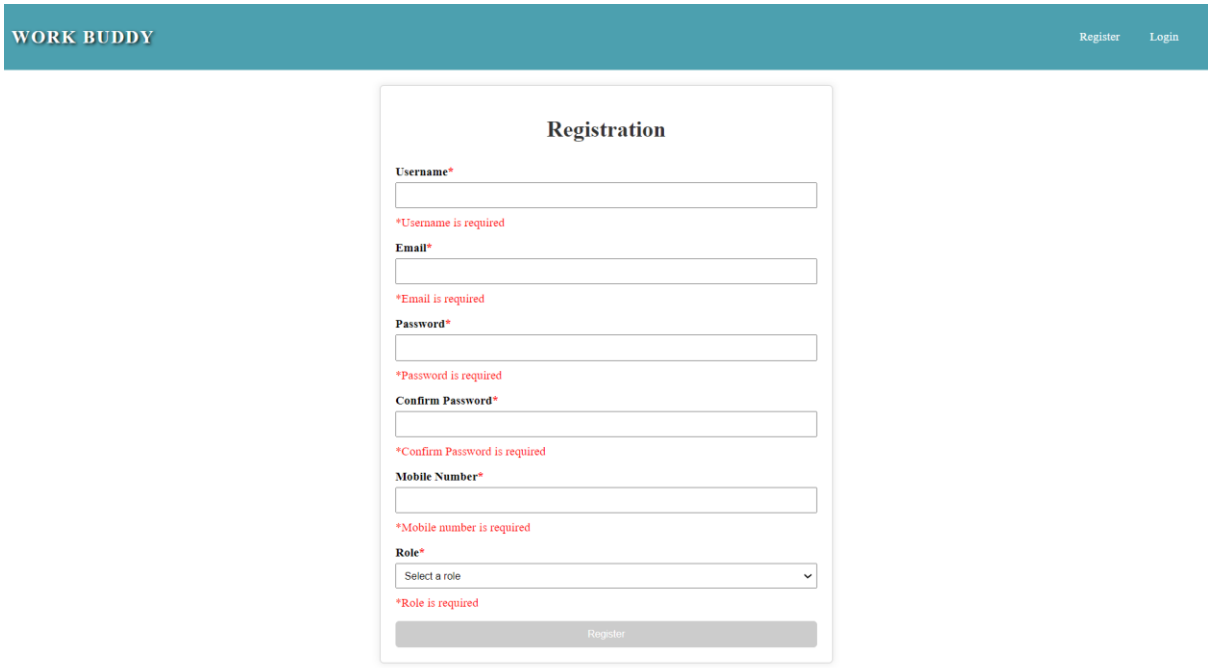
Component features a heading "**WorkBuddy**" accompanied by an introductory message that provides an overview of the application**.**

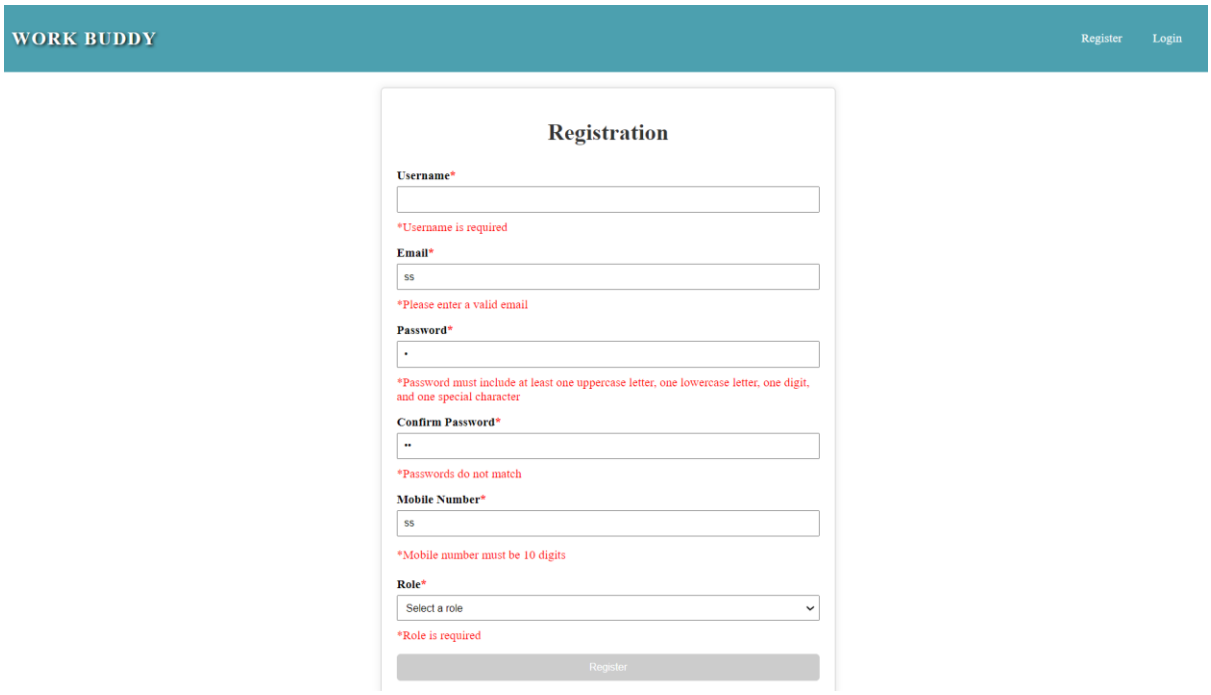
Common Screens (Employee and Manager)

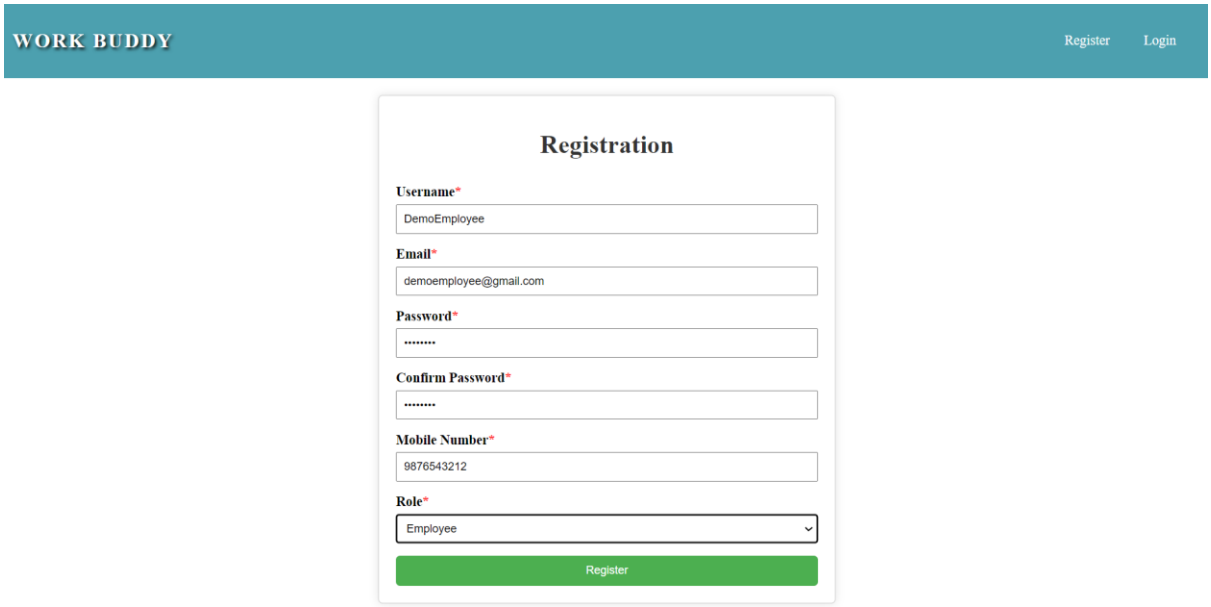
Registration Page:
(**registration component**)

Clicking "Register" in the navbar displays the registration page for both roles. Please refer to the screenshots below for validations.

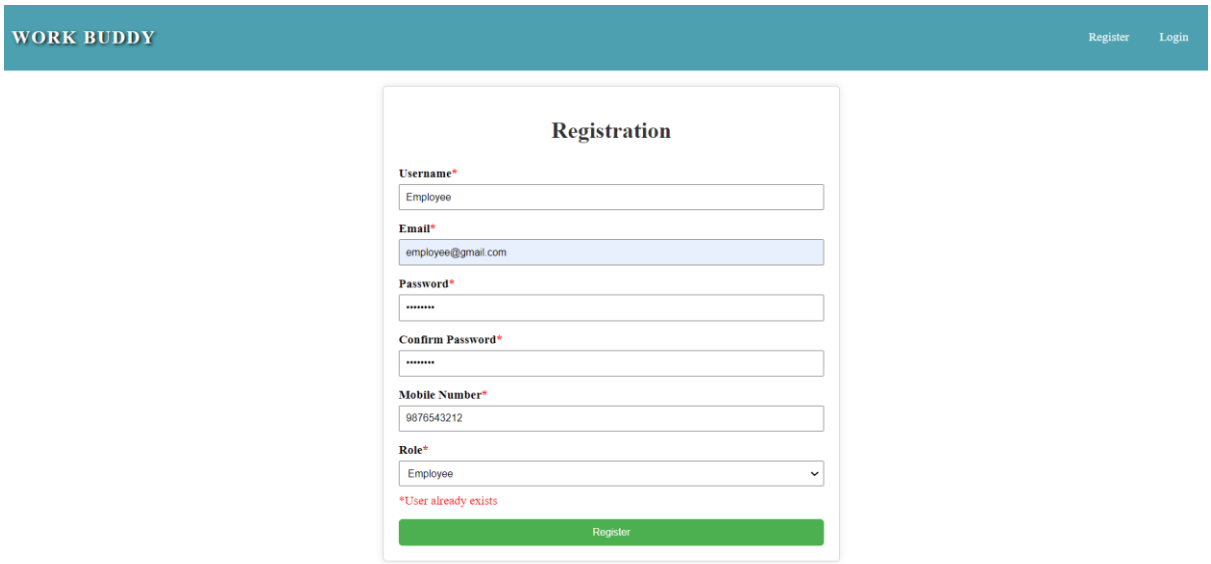



WORK BUDDY
Register
Login
Registration
Username*
*Username is required
Email*
*Email is required
Password*
*Password is required
Confirm Password*
*Confirm Password is required
Mobile Number*
*Mobile number is required
Role*
Select a role
*Role is required
Register


WORK BUDDY
Register
Login
Registration
Username*
*Username is required
Email*
ss
*Please enter a valid email
Password*
•
*Password must include at least one uppercase letter, one lowercase letter, one digit, and one special character
Confirm Password*
••
*Passwords do not match
Mobile Number*
ss
*Mobile number must be 10 digits
Role*
Select a role
*Role is required
Register



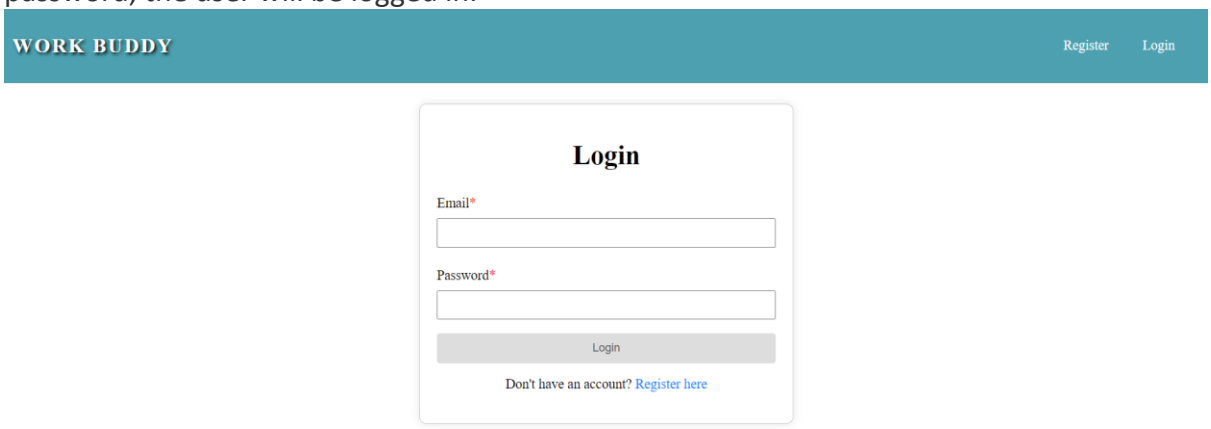
When the "Register" button is clicked, upon successful submission, the user must be navigated to the login page.



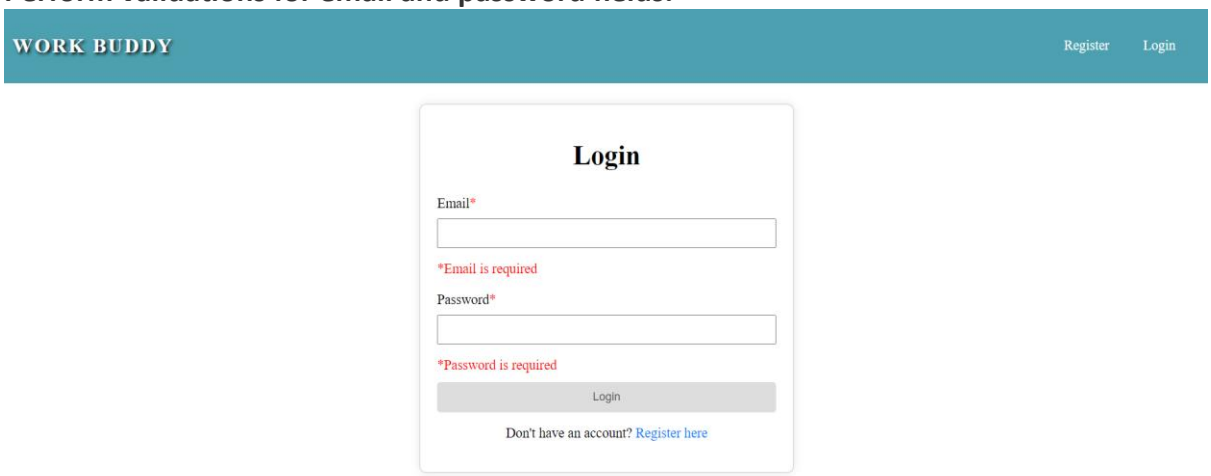
If a user attempts to register with an existing email, a message stating "User already exists" will be displayed.

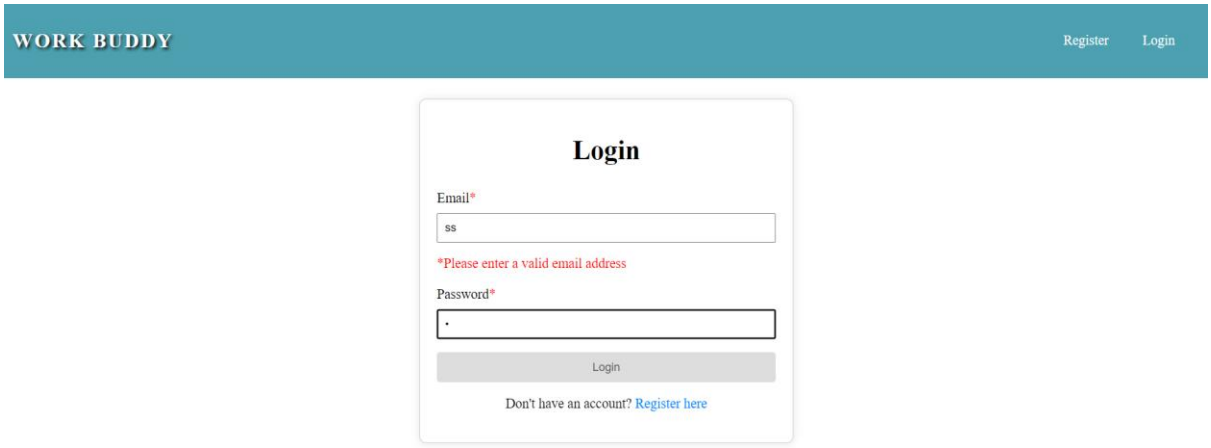
Login Page
(**login component**)

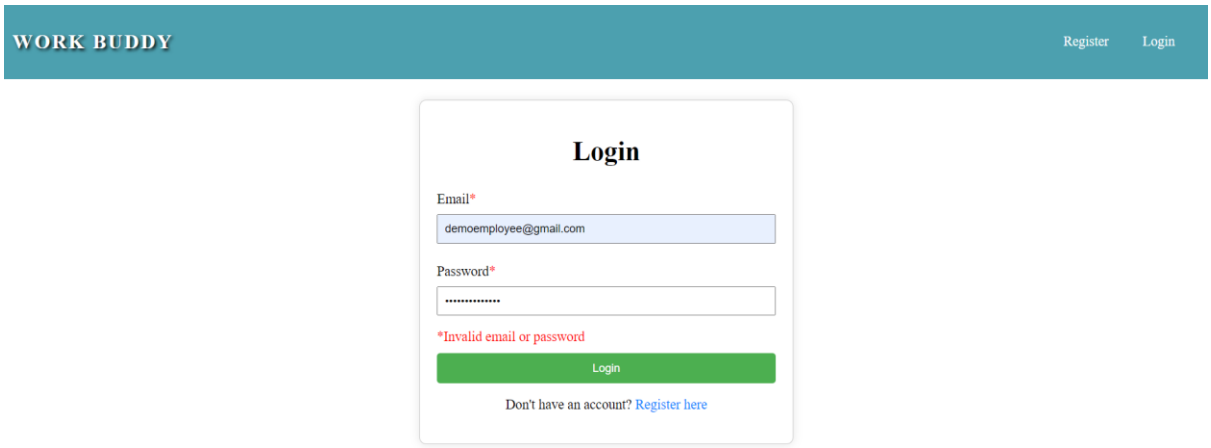
This page is used for logging in to the application. On providing the valid email and password, the user will be logged in.

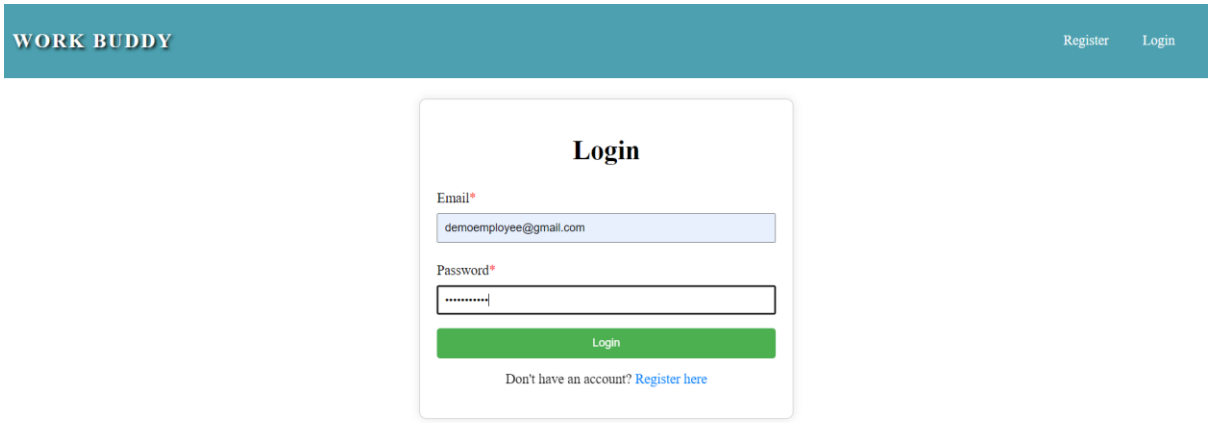


Perform validations for email and password fields.




WORK BUDDY
Register
Login
Login
Email*
ss
*Please enter a valid email address
Password*
Login
Don't have an account? Register here

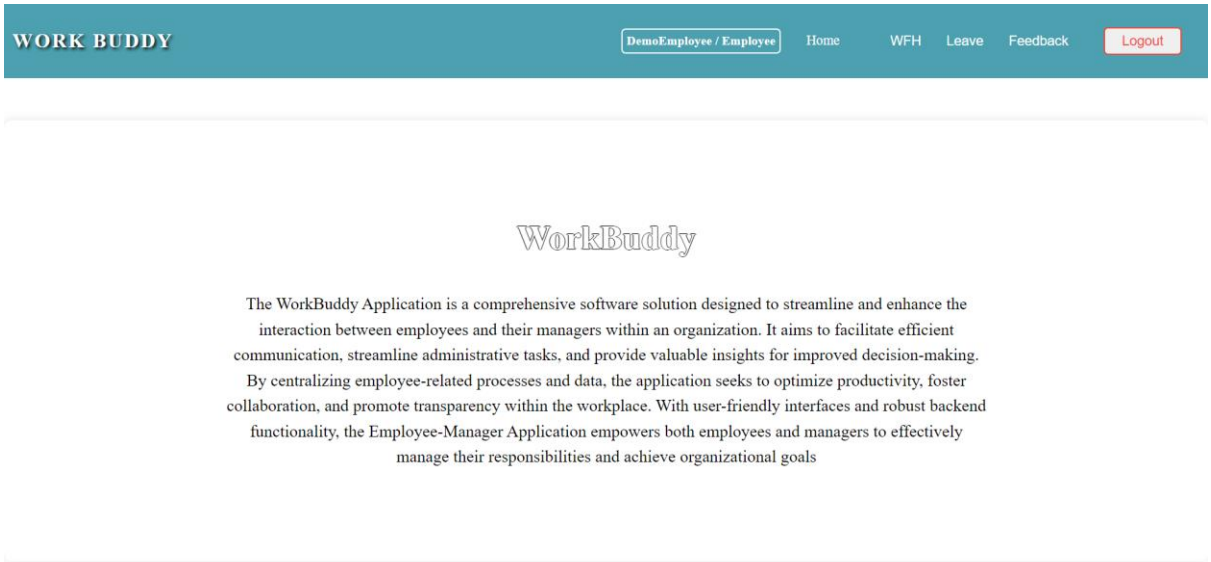

WORK BUDDY
Register
Login
Login
Email*
demoemployee@gmail.com
Password*
*Invalid email or password
Login
Don't have an account? Register here



On Clicking the ‘Login’ button, user will be navigated to the (adminnav) based on their roles.

Employee side:

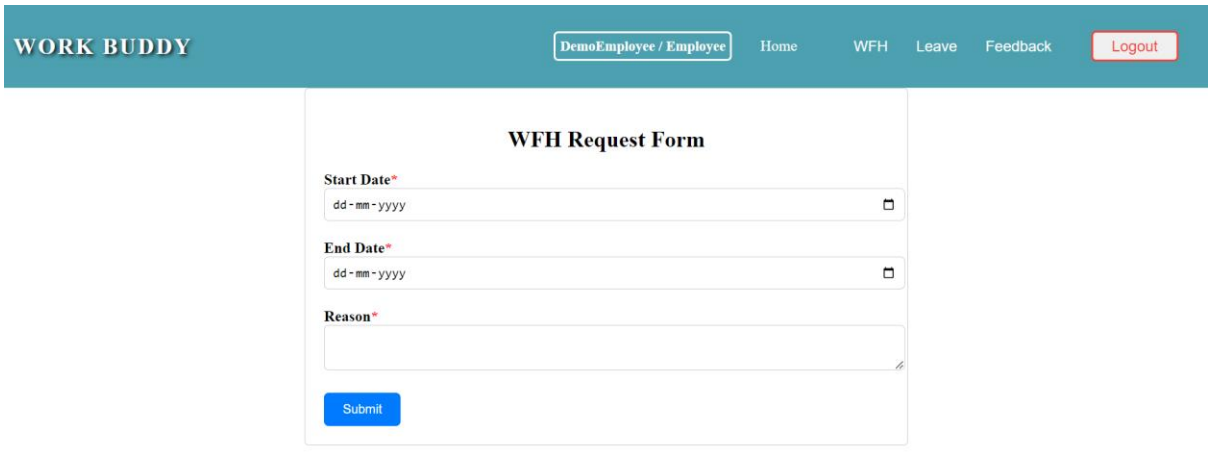
Home Component: This page is used to display the information about the work buddy. On clicking the ‘Home’ tab, user can view the information about the application.



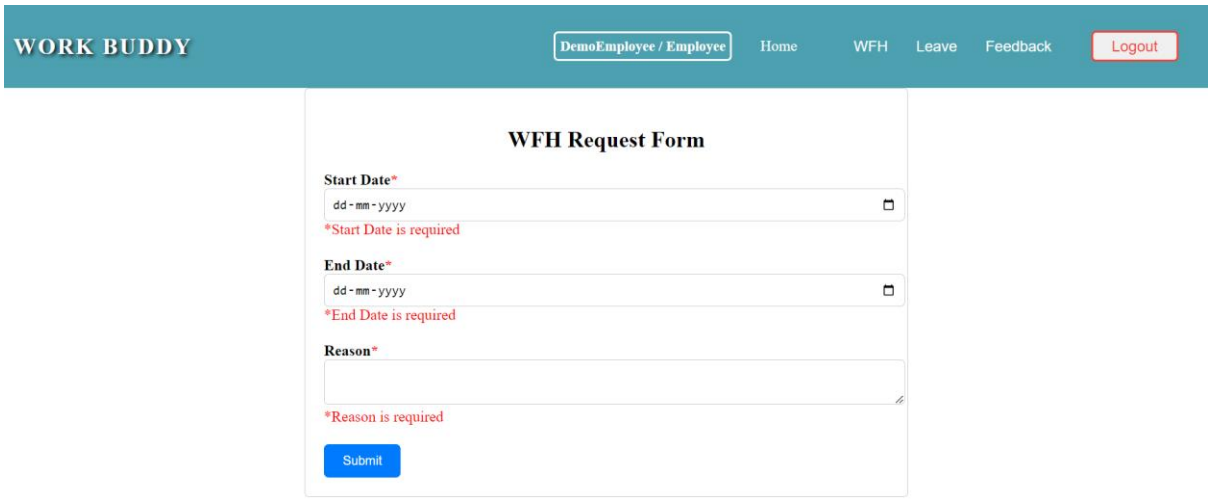
Upon successful login, if the user is an manager, the (adminnav component) will be displayed. If the user is a employee, the (usernav component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

WFH:
On hovering over the "WFH" item in the navbar, a submenu should appear with options to "Apply WFH" and "My WFH History".

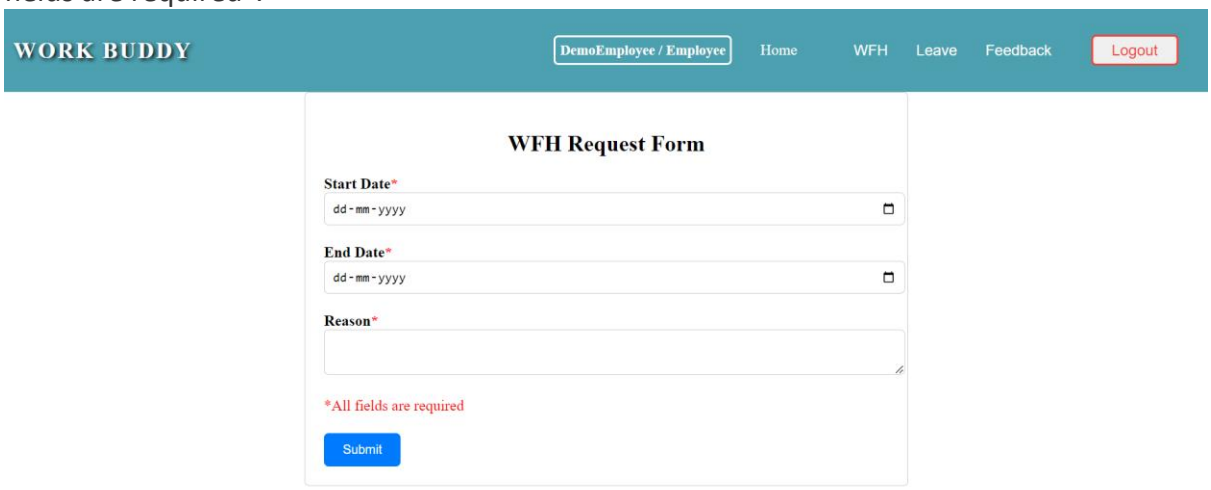
Clicking on “Apply WFH” will navigate to the “**addwfh**” component, which displays the form to apply wfh with heading as "WFH Request Form"



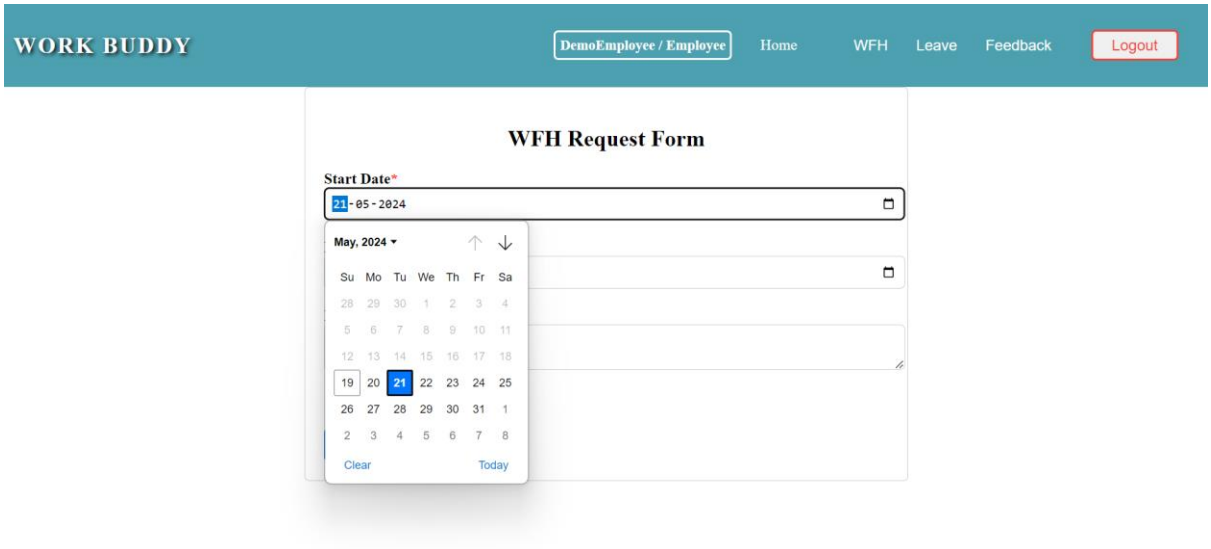
Perform validations for all the form fields.

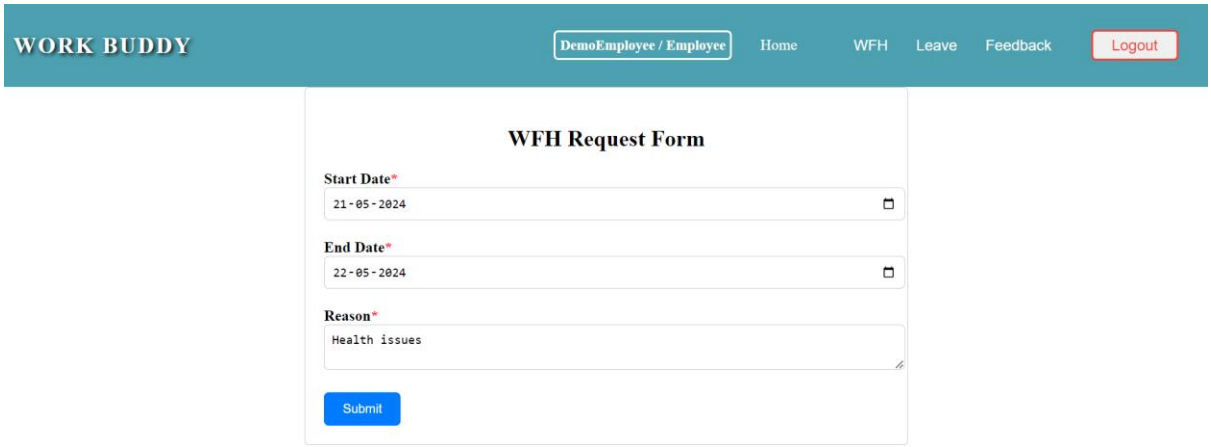


Clicking the "Submit" button with empty fields will display a validation message stating "All fields are required".

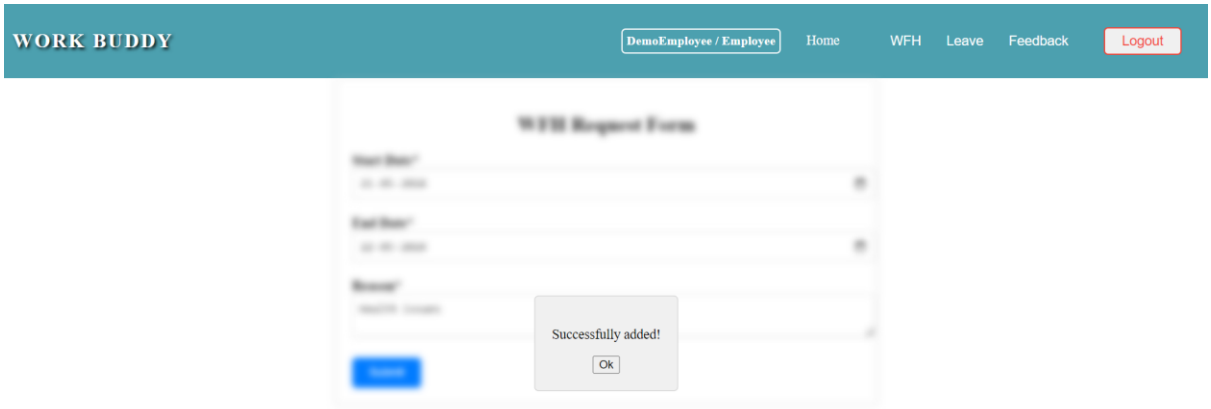


For the Dates, past dates should be disabled, allowing only future dates to be selected.

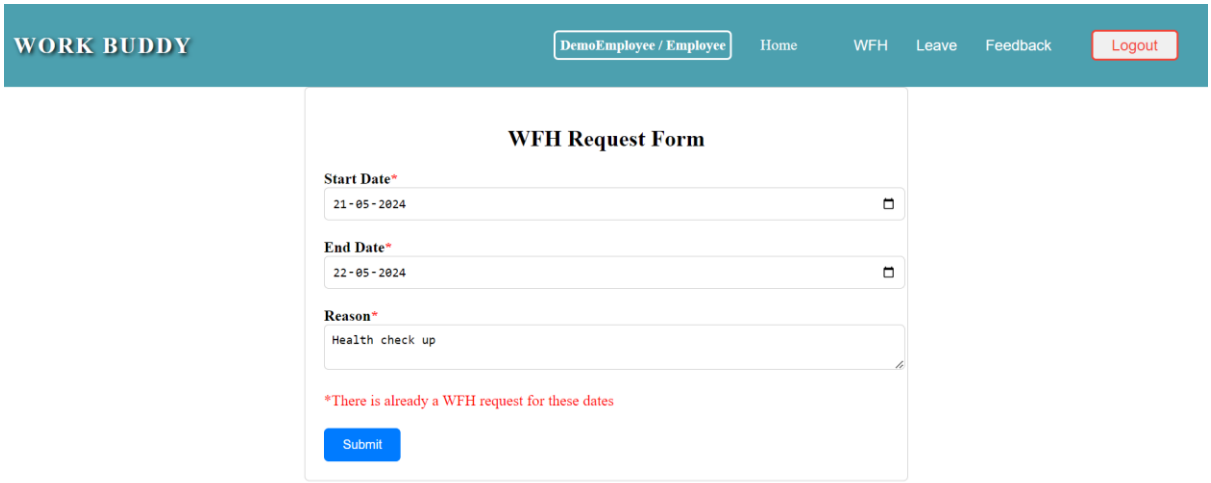




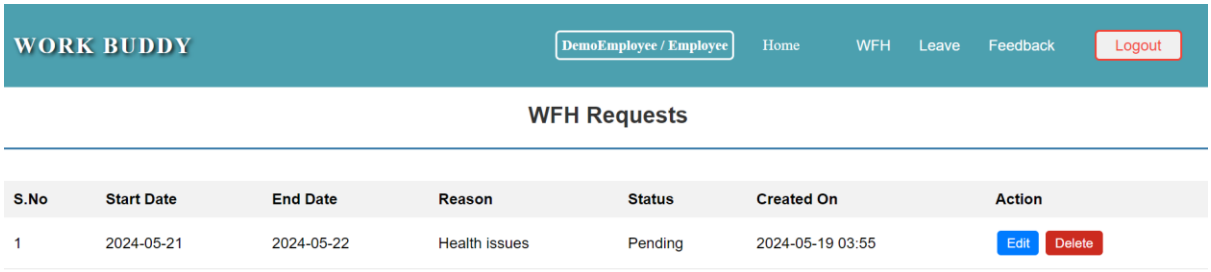
Upon clicking the "Submit" button, if the operation is successful, a popup message "Successfully Added!" should be displayed.



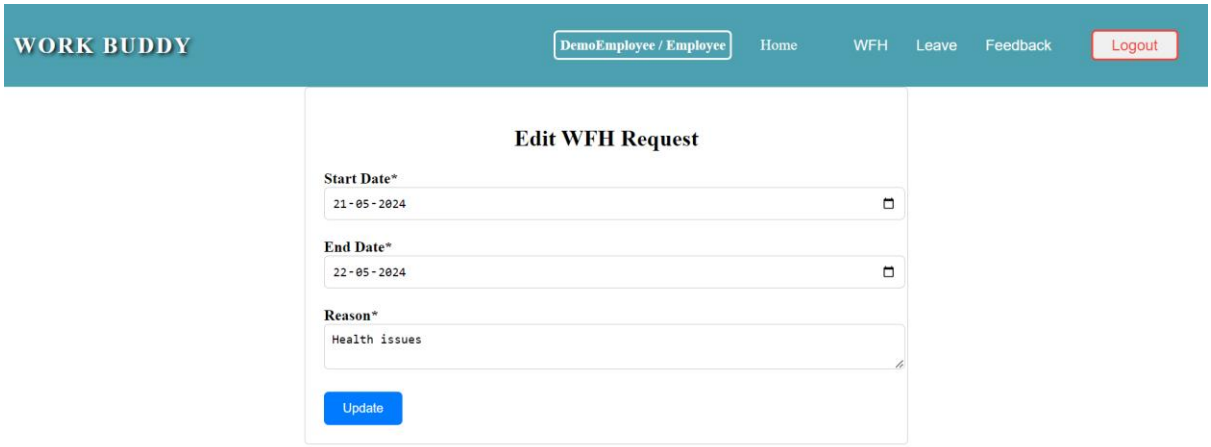
Upon clicking the "Submit" button, if any exception occurs, display the message.



Clicking on “My WFH history” will navigate to the “**mywfh**” component, which displays the form to apply wfh with heading as "WFH Requests"

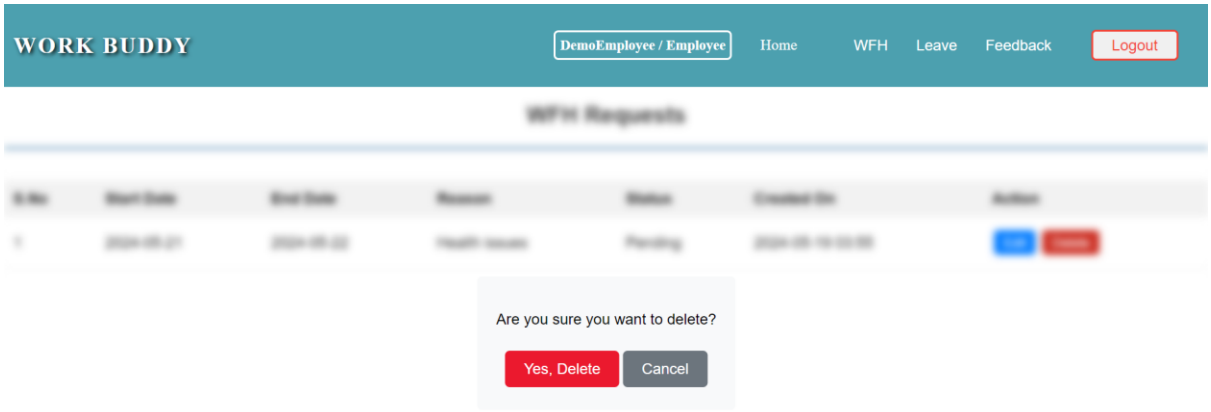


Clicking on “Edit” button will navigate to the “**addwfh**” component, where the data will be prefilled.



Clicking on “Update” button will update the wfh record.

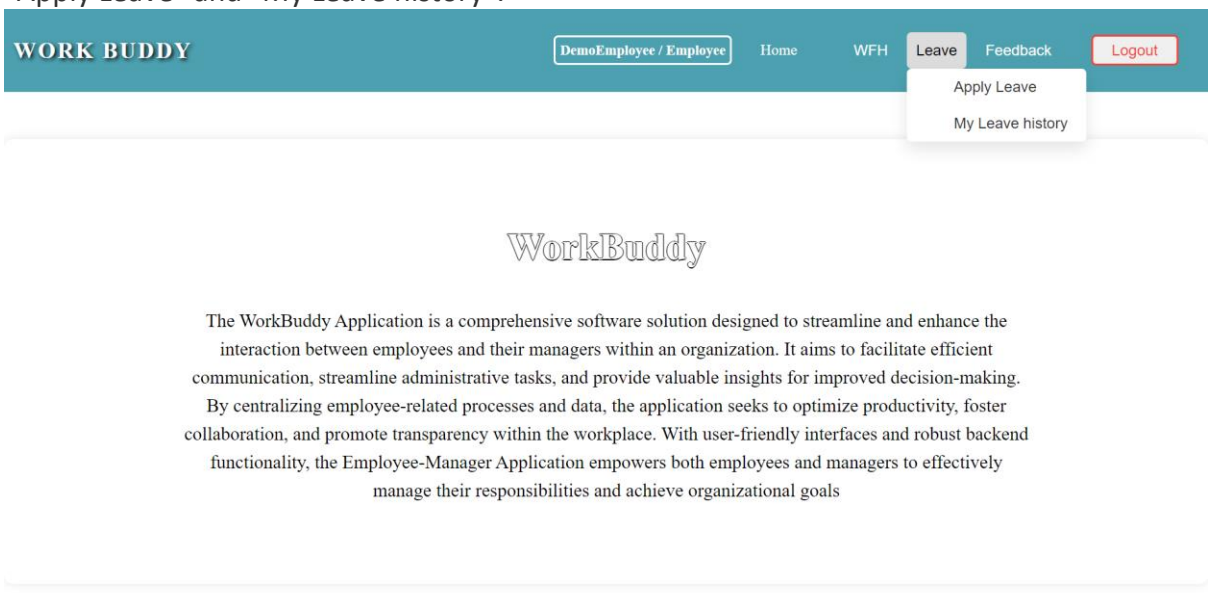
On clicking the "**Delete**" button, a pop-up should be displayed with confirmatory message to delete the data.



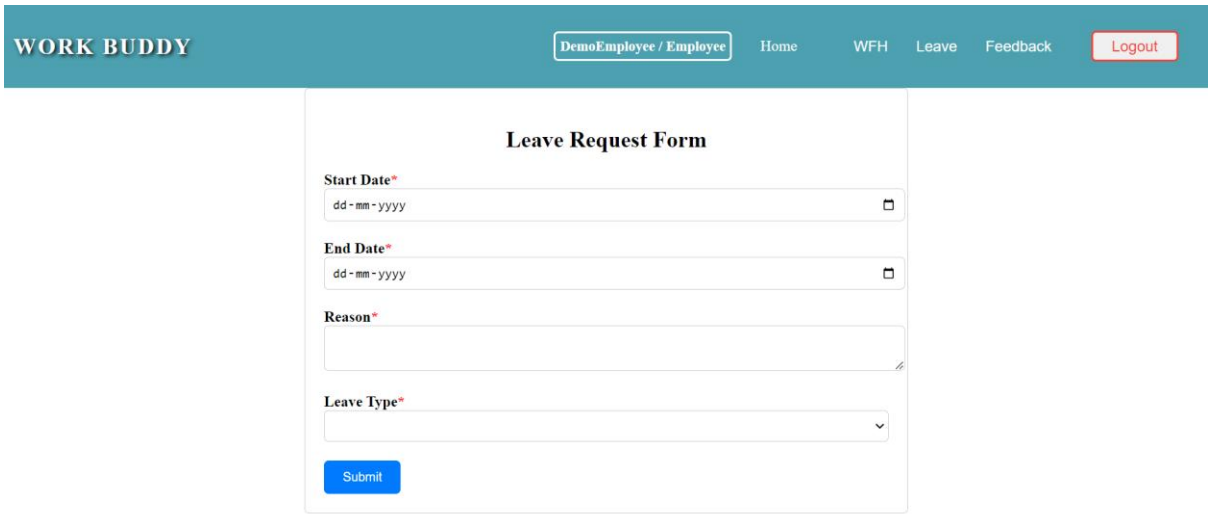
Clicking "Yes, Delete" will delete the wfh, and the change will be automatically reflected.

Leave:

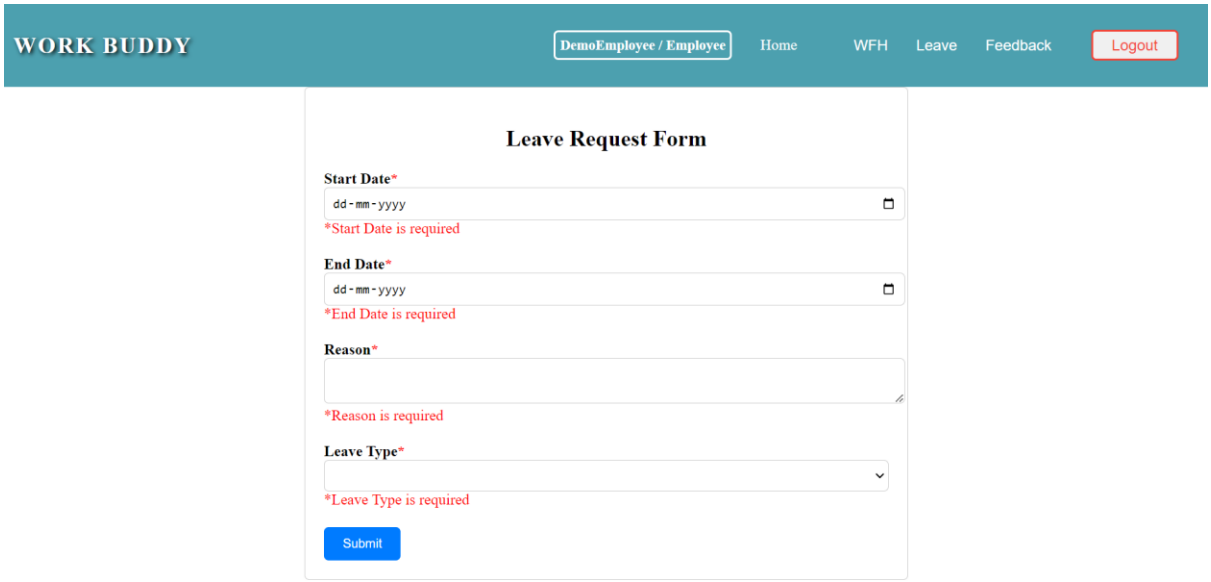
On hovering over the "Leave" item in the navbar, a submenu should appear with options to "Apply Leave" and "My Leave history".



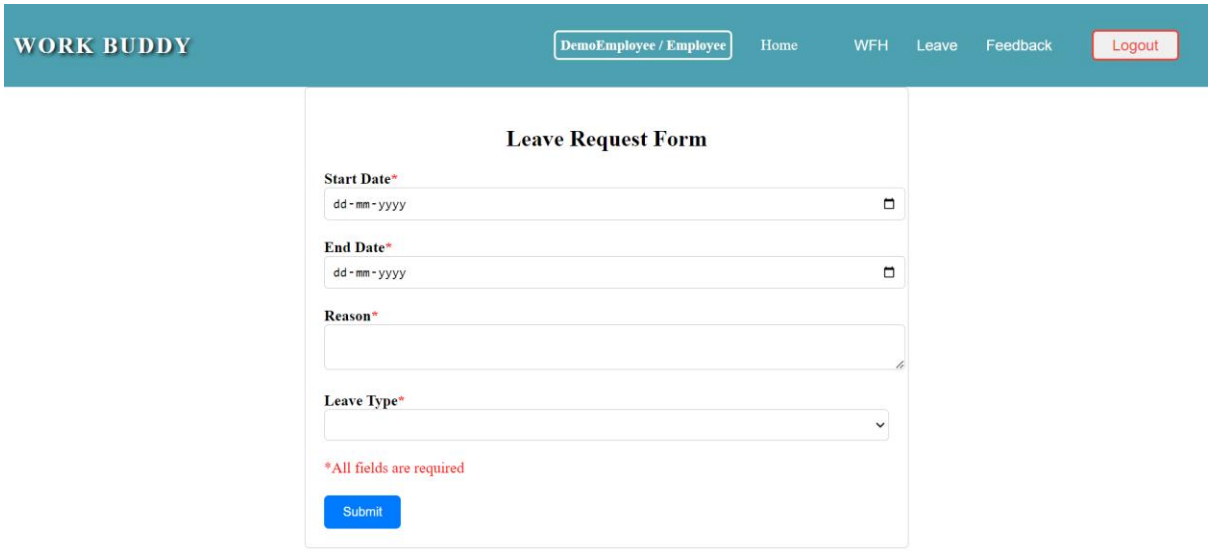
Clicking on “Apply Leave” will navigate to the “**addleave**” component, which displays the form to apply leave with heading as “**Leave Request Form**"



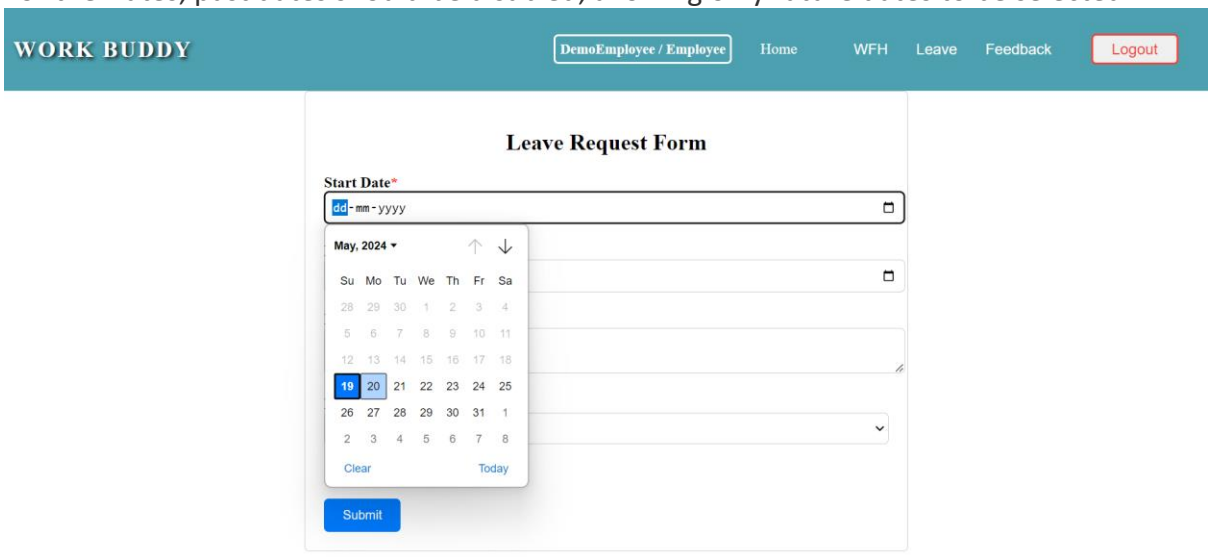
Perform validations for all the form fields.

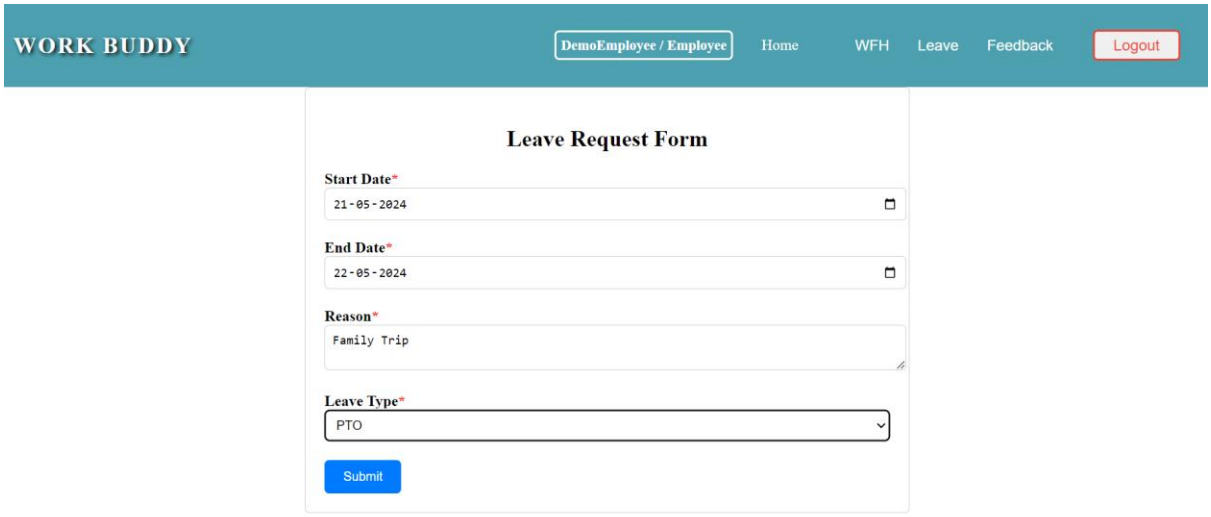


Clicking the "Submit" button with empty fields will display a validation message stating, "All fields are required".

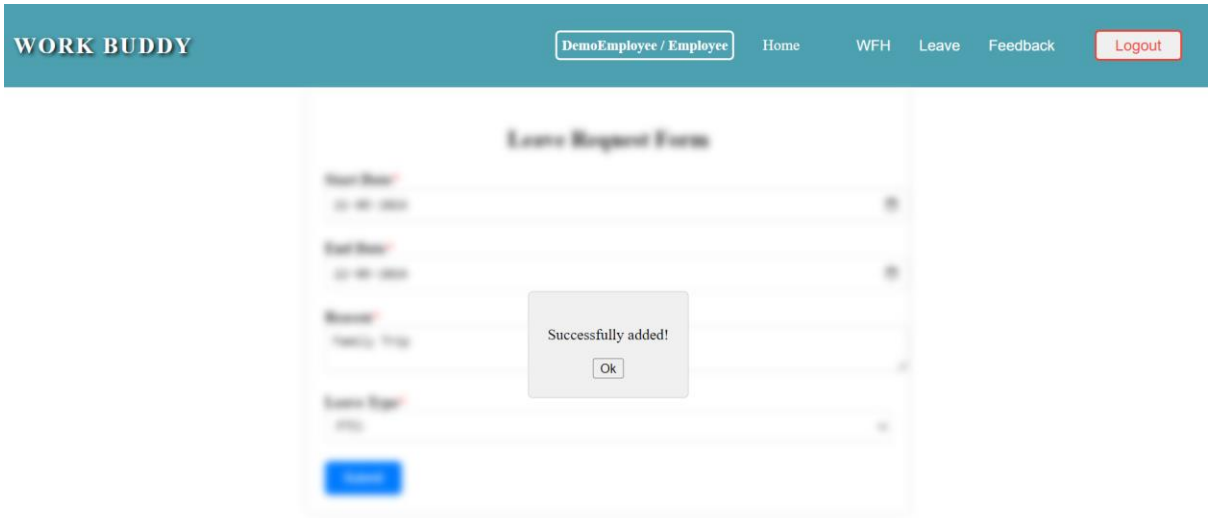


For the Dates, past dates should be disabled, allowing only future dates to be selected.

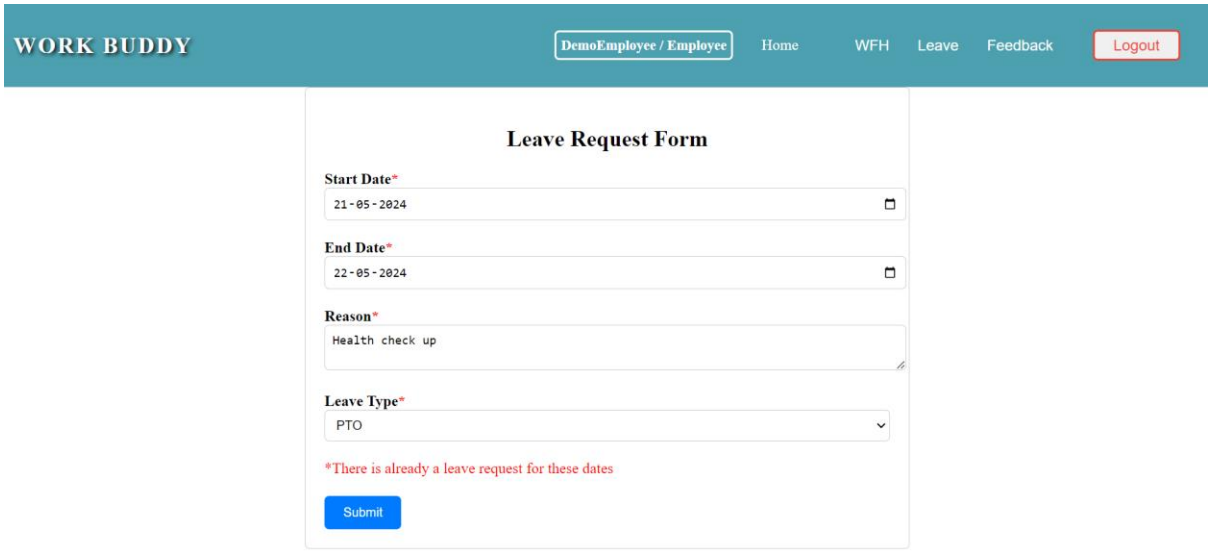




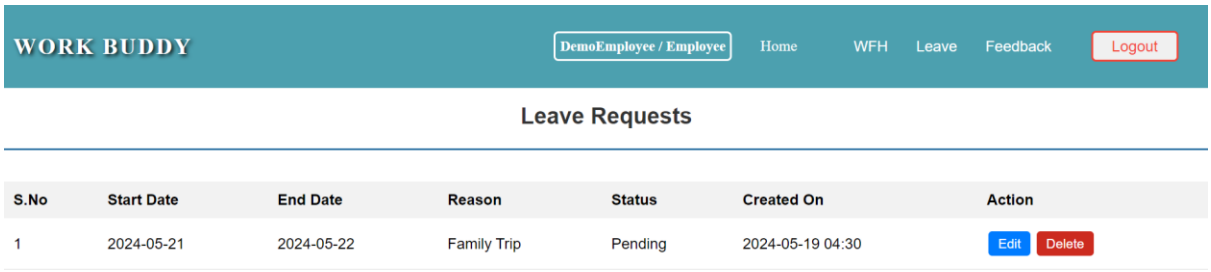
Upon clicking the "Submit" button, if the operation is successful, a popup message "Successfully Added!" should be displayed.



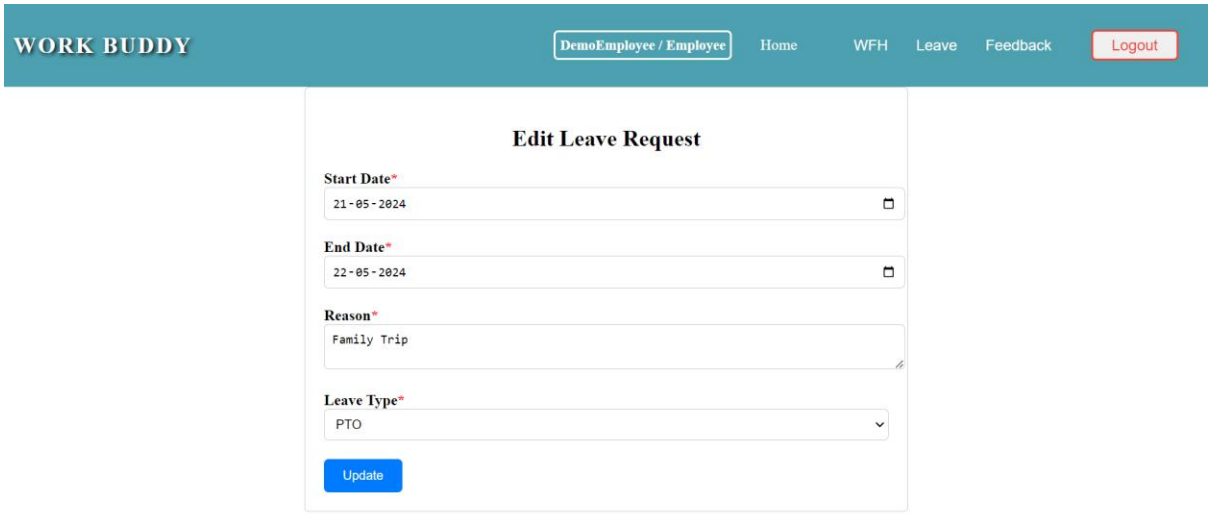
Upon clicking the "Submit" button, if any exception occurs, display the message.



Clicking on “My leave history” will navigate to the “**myleave**” component, which displays the form to apply leave with heading as "Leave Requests"

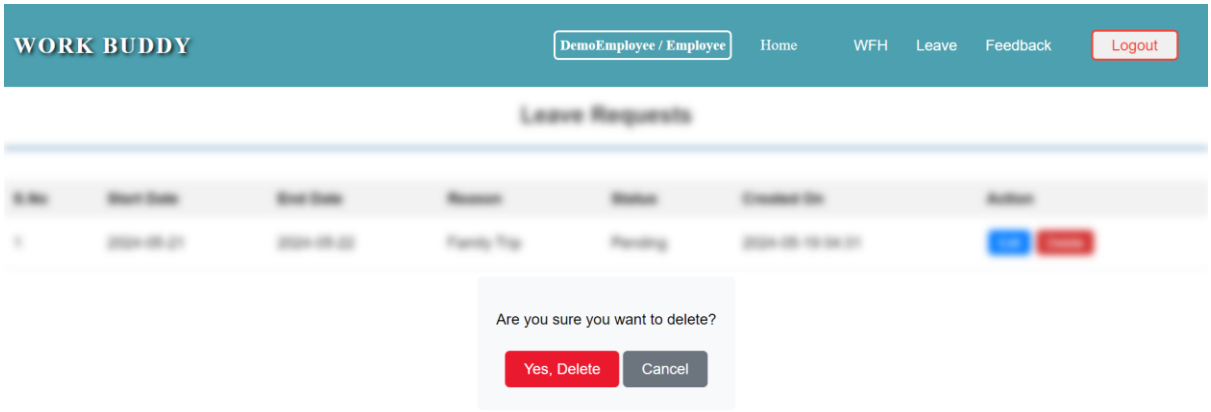


Clicking on “Edit” button will navigate to the “**addleave**” component, where the data will be prefilled.



Clicking on “Update” button will update the leave record.

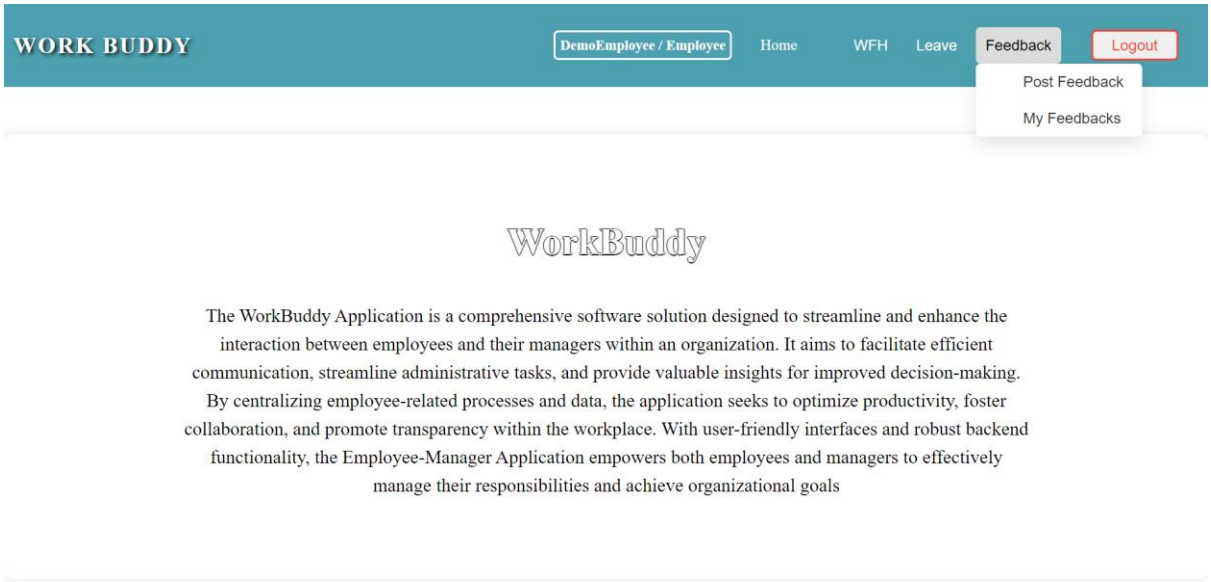
On clicking the "**Delete**" button, a pop-up should be displayed with confirmatory message to delete the data.



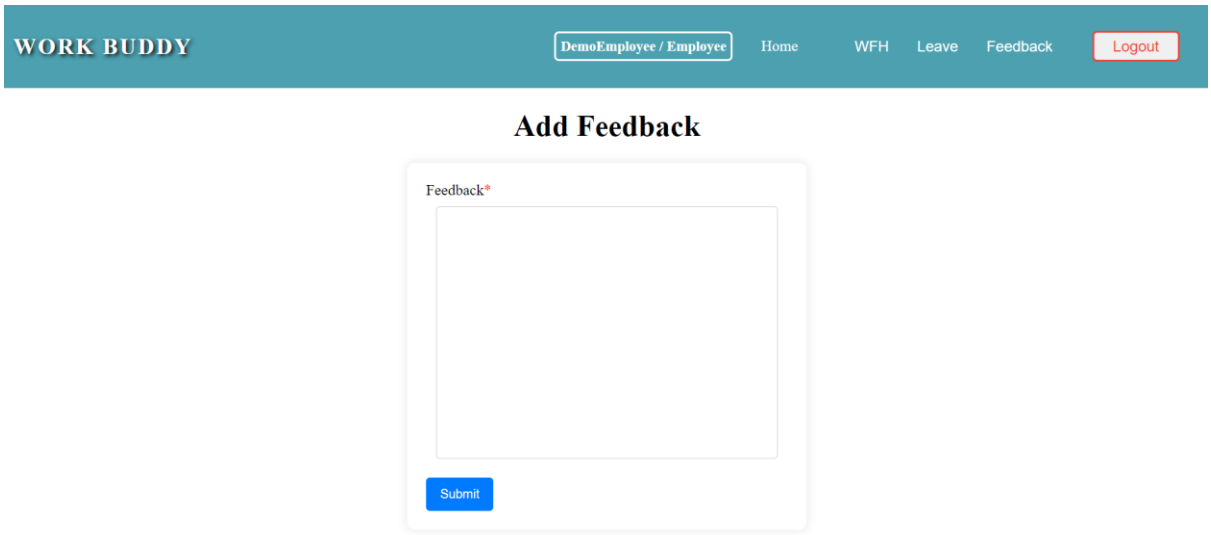
Clicking "Yes, Delete" will delete the leave, and the change will be automatically reflected.

Feedback:

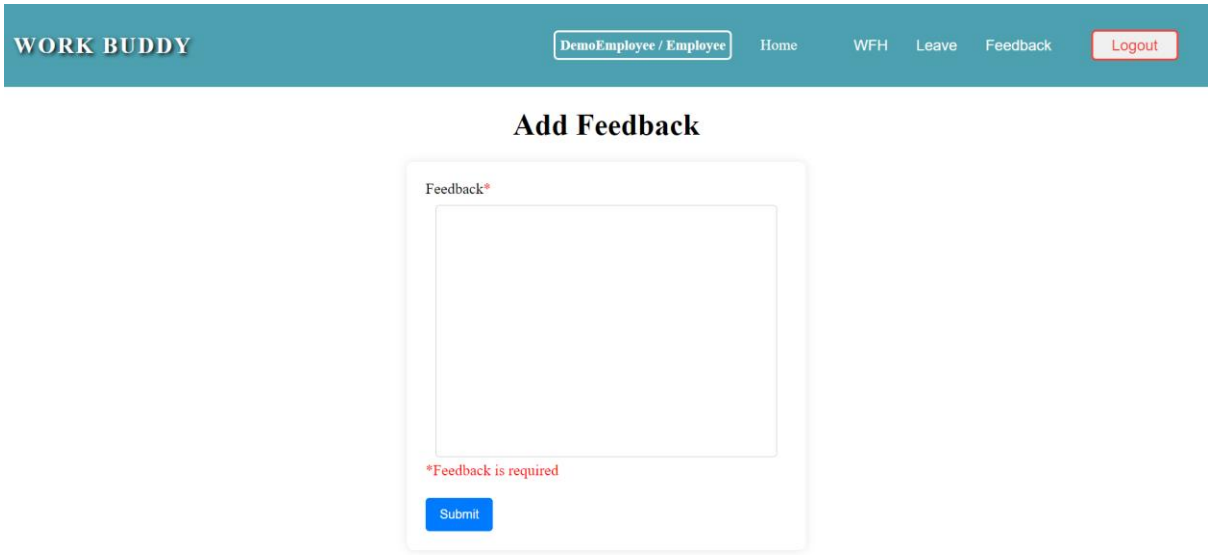
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".



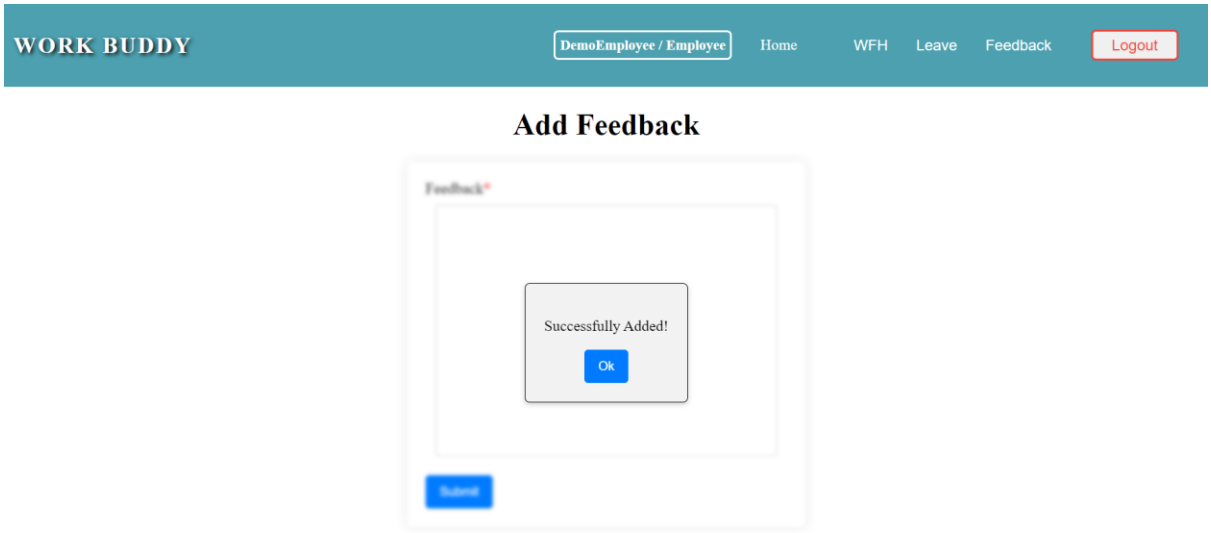
Clicking on "Post Feedback" will navigate to the **useraddfeedback** component, which displays the form to post feedback with heading as "Add Feedback"



Clicking the "Submit" button with empty textarea will display a validation message stating "Feedback is required".



Upon clicking the "Submit" button, if the operation is successful, a popup message saying "Successfully Added!" should be displayed.

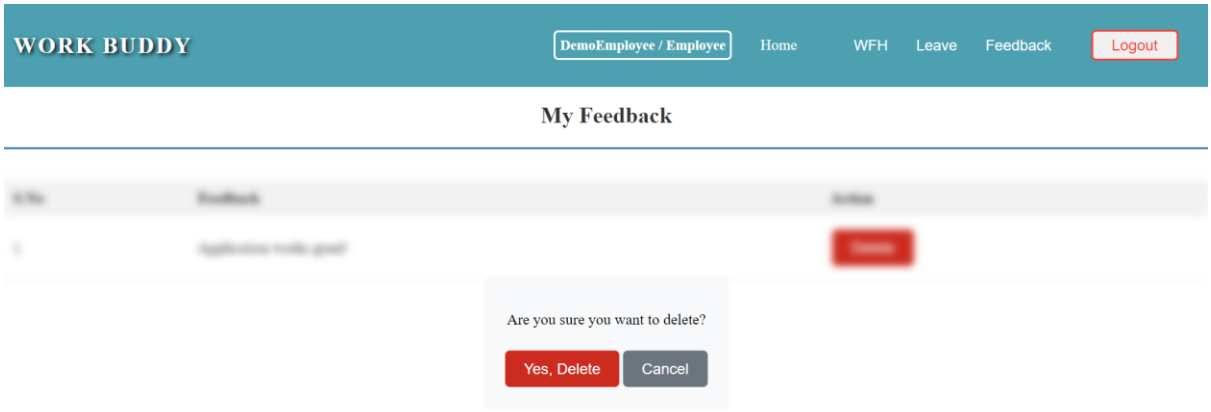


Clicking the "Ok" button will close the popup, and the same **useraddfeedback component** will be displayed.

User view Feedback:

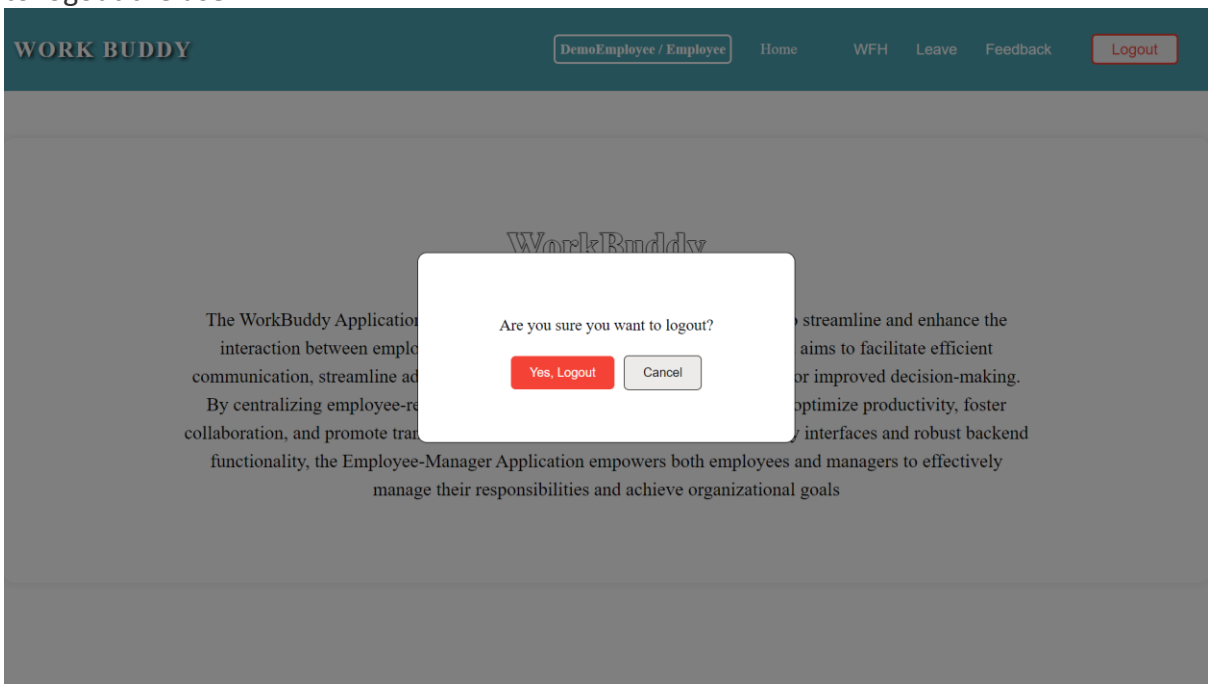
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".
Clicking on "My Feedbacks" will navigate to the **userviewfeedback component**, which displays posted feedback with heading as "My Feedback"

On clicking the "Delete" button, a pop-up should be displayed with confirmatory message to delete the data.

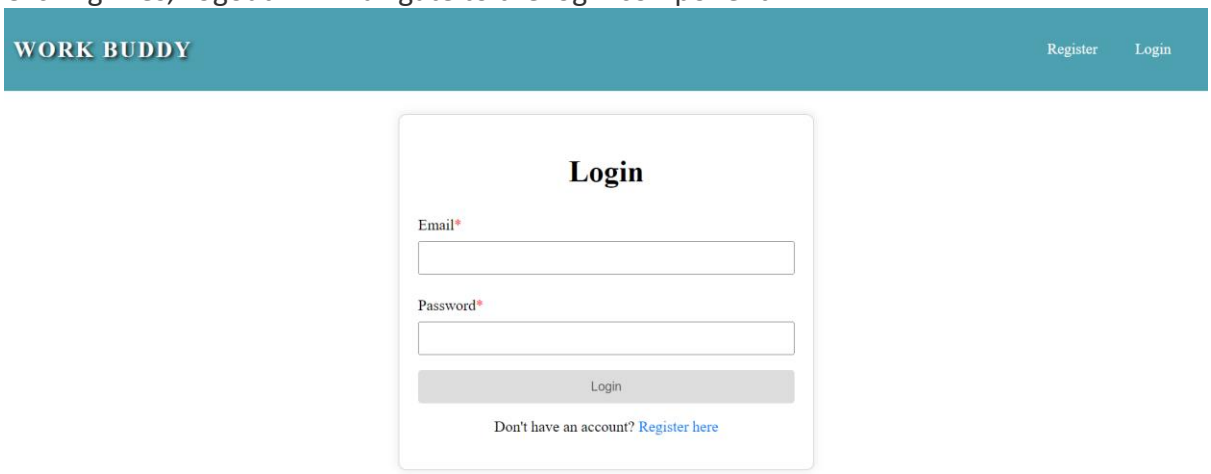


Clicking "Yes, Delete" will delete the feedback posted, and the change will be automatically reflected.

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.

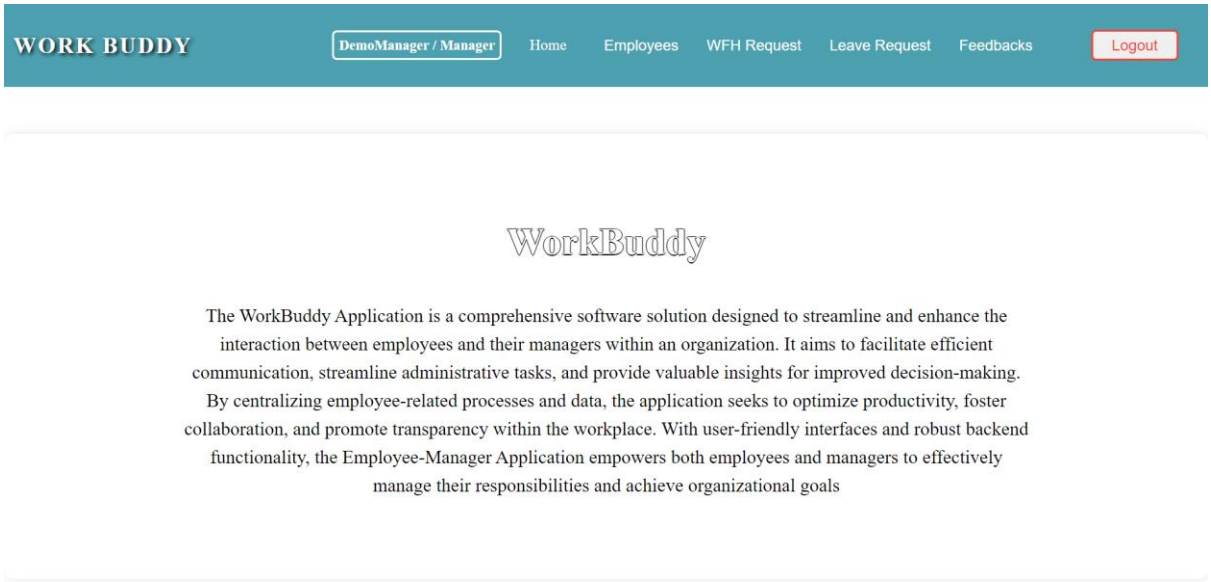


Clicking "Yes, Logout" will navigate to the login component.



<u>Manager side:</u>

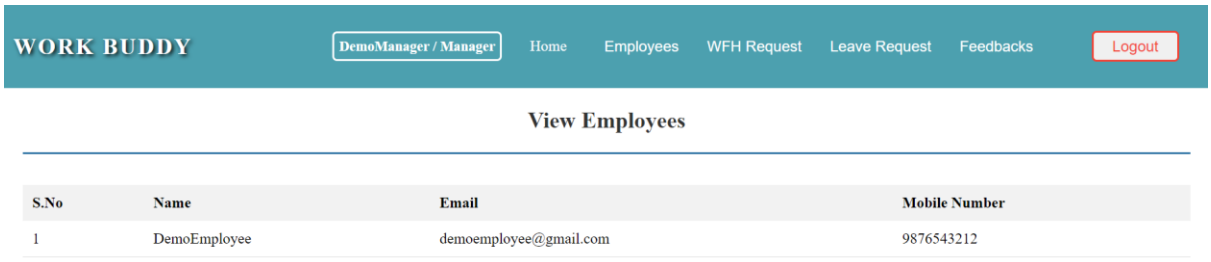
Home Component: This page is used to display the information about the WorkBuddy application. On clicking the '**Home**' tab, user can view the information about the application.



Upon successful login, if the user is an manager, the (**adminnav** component) will be displayed. If the user is a employee, the (**usernav** component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

Managers can navigate to other pages by clicking on the menu available in the navigation bar.

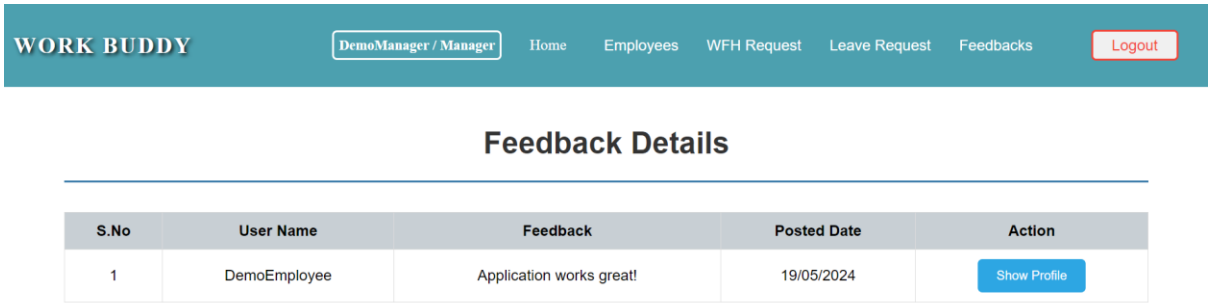
By clicking the “Employees” option will navigate to the “**viewemployees**” component.



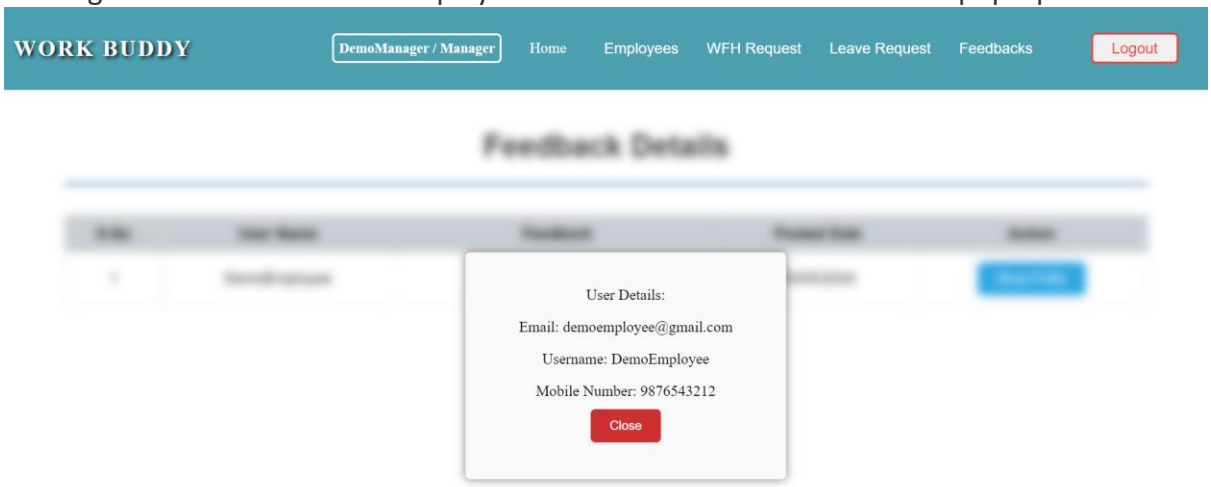
By clicking the “WFH Request” option will navigate to the “**viewwfh**” component, where the manager can approve or reject the request.

By clicking the “Leave Request” option will navigate to the “**viewleave**” component, where the manager can approve or reject the request.

Clicking on "Feedbacks" from the navbar will navigate to the **adminviewfeedback** component, which displays all feedbacks posted by all employees.

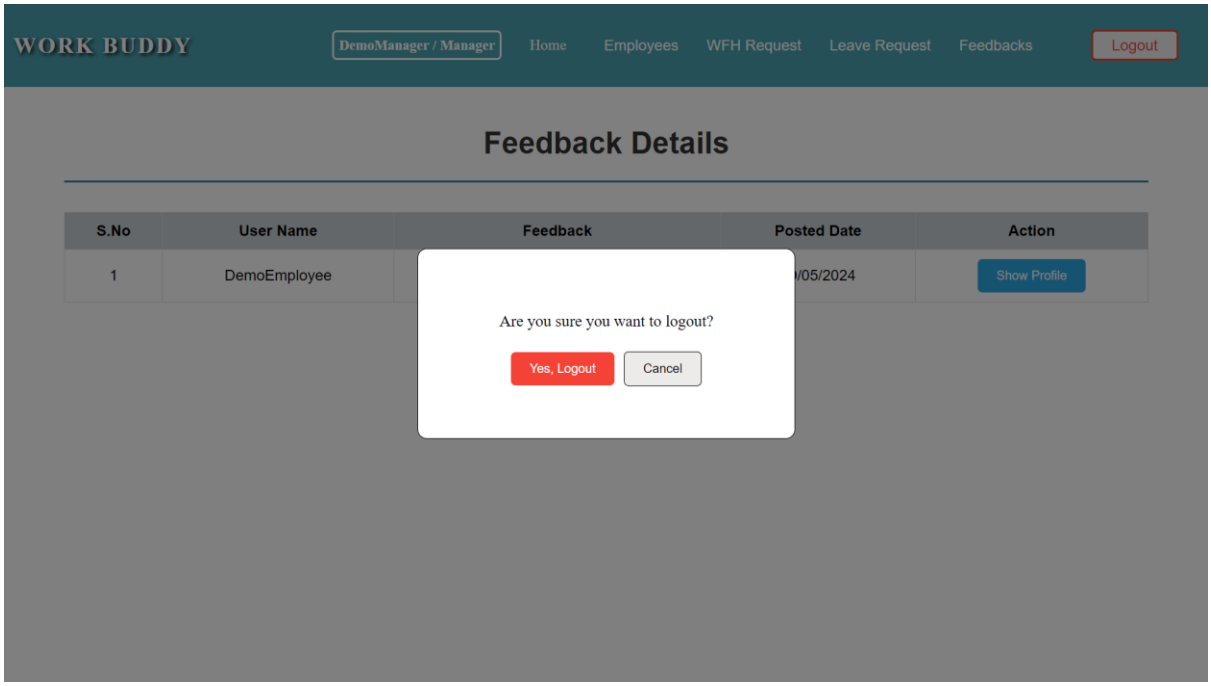


Clicking on "Show Profile" will display additional details about the user in pop-up modal.



On clicking the "Close" button, will close the pop-up modal displayed.

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.



Clicking "Yes, Logout" will navigate to the login component.