

Garden Mentor

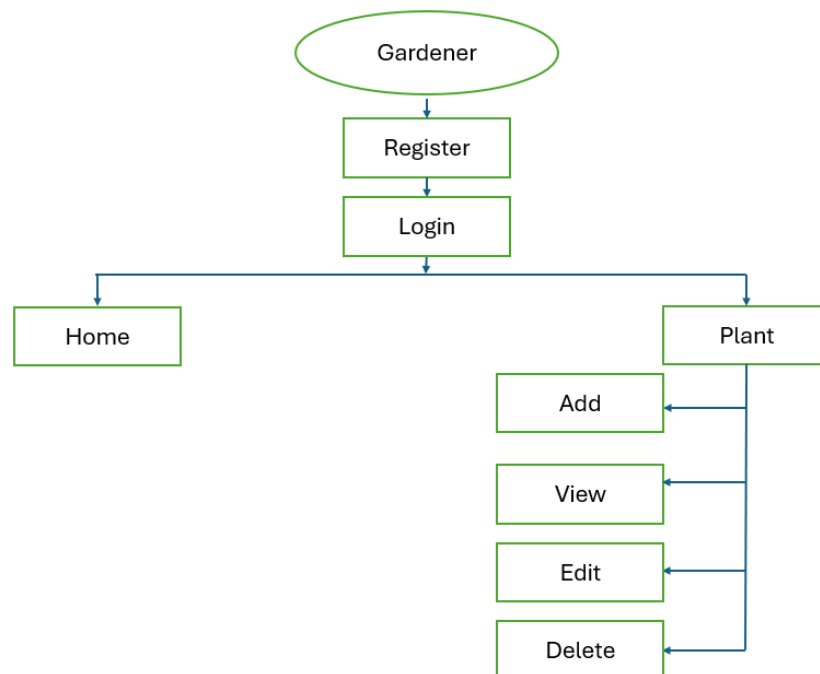
Introduction:

The Garden Mentor application is designed to assist customers in selecting the perfect plants for their spaces by providing expert recommendations and personalized gardening tips. The platform offers a user-friendly interface where users can explore curated plant collections, discover trending species, and receive tailored advice to enhance their gardening experience. Garden Mentor aims to empower users with the knowledge and guidance needed to create beautiful, thriving gardens, making the gardening journey both enjoyable and successful.

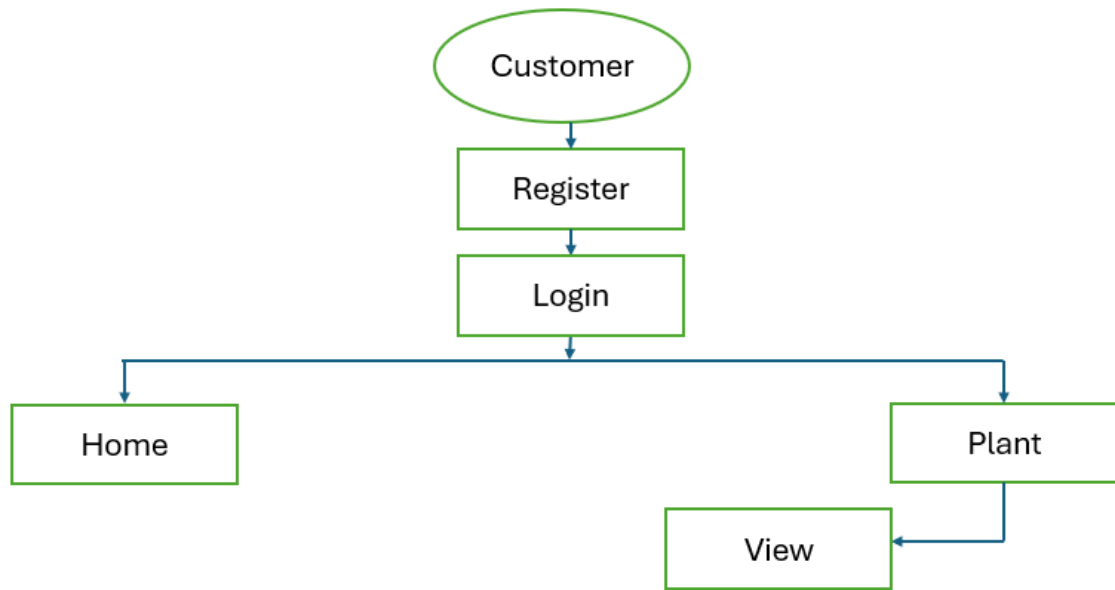
Users of the System:

1. Gardener
2. Customer

Gardener Flow Diagram:



Customer Flow Diagram:



Gardener Actions

- **Create Plants:** Gardeners create new plants by providing the name, category, price, quantity and add images for the plants.

Customer Actions

- **View Plants:** Customers can browse available plants.

Modules of the Application:

Gardener:

1. Register
2. Login
3. Home
4. Plants
 1. Add Plant
 2. View Plant
 3. Edit Plant
 4. Delete Plant

Customer:

1. Register
2. Login
3. Home
4. Plant
 1. View Plants

Technology Stack

Front End

- React

Back End

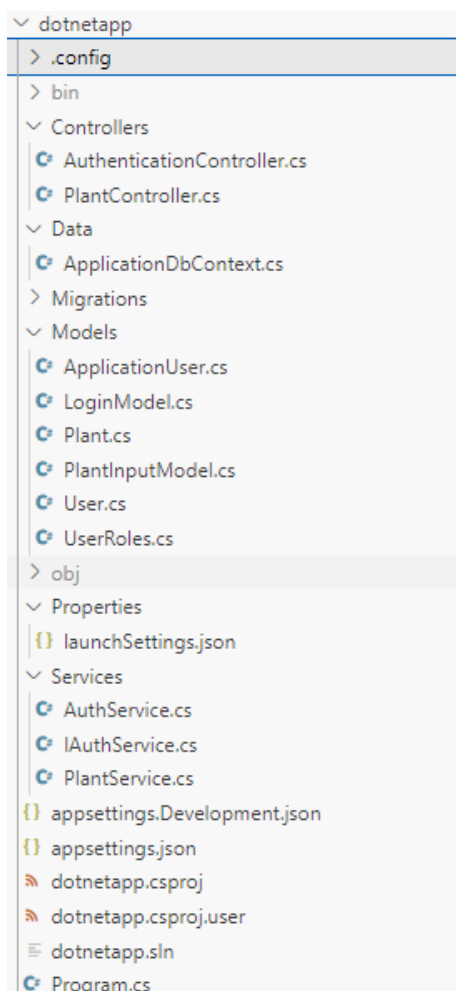
- Dotnet

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.



ApplicationDbContext: (/Data/ApplicationDbContext.cs)

Namespace: dotnetapp.Data

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

```
public DbSet<Plant> Plants { get; set; }  
public DbSet<User> Users { get; set; }
```

Model Classes:

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

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

User (Models / User.cs):

This class stores the user role (**Gardener** or **Customer**) and all user information.

Properties:

- UserId: int
- Email: string
- Password: string
- Username: string
- MobileNumber: string
- UserRole: string (**Gardener/Customer**)

Plant (Models / Plant.cs):

This class represents adding the plant by Gardener.

Properties:

- PlantId: int
- Name: string
- Category: string
- Price: decimal
- Tips: string
- PlantImage : string

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. Gardener: string
2. Customer: string

ApplicationUser (Models / ApplicationUser.cs):

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

Property:

- Name: string (Max length 30)

Services:

Inside “**Services**” folder, create all the services file mentioned below.

Namespace: All the services files should locate within the **dotnetapp.Services** namespace.

PlantService (Services / PlantService.cs):

The Plantservice class provides CRUD operations (Create, Read, Update, Delete) for managing plant entities in the application, ensuring proper handling and validation of plant data within the context of an ApplicationDbContext. It simplifies and abstracts the interaction with the database, offering a centralized service for plant sales.

Constructor:

```
public PlantService (ApplicationDbContext context)

{

    _context = context;

}
```

Functions:

1. **public async Task<IEnumerable<Plant>> GetAllPlants():**
 - Retrieves all plants from the database.
2. **public async Task<Plant> GetPlantById(int plantId):**
 - Retrieves a Plant from the database with the specified PlantId.
3. **public async Task<bool> AddPlant(Plant plant):**
 - Adds a new plant to the database.
 - Checks if a plant with the same name already exists.
 - If no such plant exists, adds the new plant to the database.

- Saves changes asynchronously to the database.
- Returns true for a successful insertion.

4. **public async Task<bool> UpdatePlant(int plantId, Plant plant):**

- Updates an existing Plant in the database with the values from the provided plant object.
- If no plant with the specified PlantId is found, returns false.
- Before updating, it checks if another plant with the same category already exists for the same user. If a plant with an overlapping category is found, the update is not performed, and it returns false.
- If no overlap is detected, saves changes asynchronously to the database.
- Returns true for a successful update.

5. **public async Task<bool> DeletePlant(int plantId):**

- Deletes a plant from the database with the specified PlantId.
- Retrieves the plant based on the provided PlantId.
- If no plant with the specified PlantId is found, returns false.
- If the plant is found, deletes it from the database.
- Saves changes asynchronously to the database.
- Returns true for a successful deletion.

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, object)> 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.

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, object)> Login (LoginModel model);

Controllers:

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

Namespace: All the controllers files should locate within the **dotnetapp.Controllers** namespace.

AuthenticationController (Controllers / AuthenticationController.cs):

This controller handles user authentication and registration requests.

Functions:

1. public async Task<ActionResult> 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 **201 Created 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<ActionResult> 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 **201 Created 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.

PlantController (Controllers / PlantController.cs):

This controller manages Plants, interacting with the **PlantService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<Plant>>> GetAllPlants():

- a) The **GetAllPlants** method is a controller action responsible for retrieving all plants.
- b) Requires authorization to access.

- c) Calls the **_PlantService.GetAllPlants()** method to fetch all plants from the service layer.
- d) Returns a 200 OK response with the retrieved plants upon successful retrieval.

2. public async Task<ActionResult< Plant >> GetPlantById(int plantId):

- a) The **GetPlantById** method is a controller action responsible for retrieving a plant by its ID.
- b) Calls the **_PlantService.GetPlantById(PlantId)** method to retrieve the plant from the service layer.
- c) If the plant is not found, returns a 404 Not Found response with a message "Cannot find any plant".
- d) If the plant is found, returns a 200 OK response with the plant data.

3. public async Task<ActionResult> AddPlant([FromBody] Plant plant):

- a) The **AddPlant** method is a controller action responsible for adding a new plant.
- b) Implements the logic inside a try-catch block.
- c) Receives the plant data in the request body.
- d) Tries to add the plant using the **_PlantService.AddPlant(plant)** with a success message "Plant added successfully".
- e) If adding the plant fails, returns a 500 Internal Server Error response with a failure message "Failed to add plant".
- f) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdatePlant (int plantId, [FromBody] Plant plant):

- a) The **UpdatePlant** method is a controller action responsible for updating an existing plant.
- b) Implements the logic inside a try-catch block.
- c) Receives the plant ID and updated plant data in the request body.
- d) Tries to update the plant using the **_PlantService.UpdatePlant(PlantId, plant)** method.
- e) If the update is successful, returns a 200 OK response with a success message "Plant updated successfully".
- f) If the plant is not found, returns a 404 Not Found response with a message "Cannot find any plant".
- g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> DeletePlant (int plantId):

- a) The **DeletePlant** method is a controller action responsible for deleting a plant.
- b) Implements the logic inside a try-catch block.
- c) Receives the Plant ID to be deleted.
- d) Tries to delete the plant using **the _PlantService.DeletePlant(plantId)** method.
- e) If the deletion is successful, returns a 200 OK response with a success message "Plant deleted successfully".
- f) If the plant is not found, returns a 404 Not Found response with a message "Cannot find any plant".
- g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

Endpoints:

Authentication		^
POST	/api/login	v
POST	/api/register	v
Plant		^
GET	/api/plants	v
POST	/api/plants	v
GET	/api/plants/{plantId}	v
PUT	/api/plants/{plantId}	v
DELETE	/api/plants/{plantId}	v

Note:

1. Ensure JWT authentication is applied to all endpoints, with role-based access control:
 - The endpoint for adding a plant requires a valid JWT in the header. If not provided or invalid, return an Unauthorized error.
 - Only users with the "Gardener" role can access this endpoint. If a user with the "Customer" role (or any other unauthorized role) attempts access, return a Forbidden error.
2. If there is any unused Endpoints, you can use it for a future use (There will be no testcase).

1. Login: [Access for both Customer and Gardener]

Endpoint name: "/api/login"

Method: POST

Request body: {
 "Email": "string",
 "Password": "string"
}

Response:

Status Code	Response body
201 (HttpStatusCode OK)	JSON object containing a JWT token. Example: { "Status": "Success", "token": "eyJhbGciOiJIUzI1NiQWRtaW4iLCJqdGkiOiJmZTUyYzAxZS1"
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

2. Register: [Access for both Customer and Gardener]

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
201 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

3. Get all Plants: [Access for both Gardener and Customer]

Endpoint name: “/api/plants”

Method: GET

Response:

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

4. Get Plant by id: [Access for **Gardener**]

Endpoint name: “/api/plants/{plantId}”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing details of a plant.
500	JSON object containing Error message.

5. Add Plant: [Access for only Gardener]

Endpoint name: “/api/plants”

Method: POST

Request body:

```
{
  "PlantId": 0,
  "Name": "string",
  "Category": "string",
  "Price": 0,
  "PlantImage": "string",
  "Tips": "string"
}
```

Response:

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

6. Update Plant: [Access only for Gardener]

Endpoint name: “/api/plants/{plantId}”

Method: PUT

Parameter: plantId

Request body:

```
{
  "PlantId": 0,
  "Name": "string",
  "Category": "string",
  "Price": 0,
  "PlantImage": "string",
  "Tips": "string"
}
```

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.

7. Delete Plant: [Access for only Gardener]

Endpoint name: “/api/Plant/{plantId}”

Method: DELETE

Parameter: PlantId

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.

Configure Azure Environment and Application Settings

Step 1: Set Azure Environment Variables

Declare the given Azure Credentials in the cred.sh file.

- AZURE_CLIENT_ID
- AZURE_CLIENT_SECRET
- AZURE_TENANT_ID
- AZURE_SUBSCRIPTION_ID

Set the environment variables for the Azure client by running the following commands:

- cd dotnetapp
- source cred.sh

Step 2: Update appsettings.json

In the appsettings.json file, declare the following values to configure Key Vault and JWT settings:

```
{
  "KeyVaultConfiguration": {
    "URL": "replace with your keyvault url",
    "ConnectionStringSecretName": " replace with your connect secret name",
    "BlobSecret": " replace with your blob secret"
  },
  "BlobContainerName": " replace with your blob container name",
  "JWT": {
    "ValidAudience": " replace with your valid audience",
    "ValidIssuer": " replace with your valid issuer",
    "Secret": "replace with your secret"
  }
}
```

Step 3: Implement **DefaultAzureCredential** in program.cs file

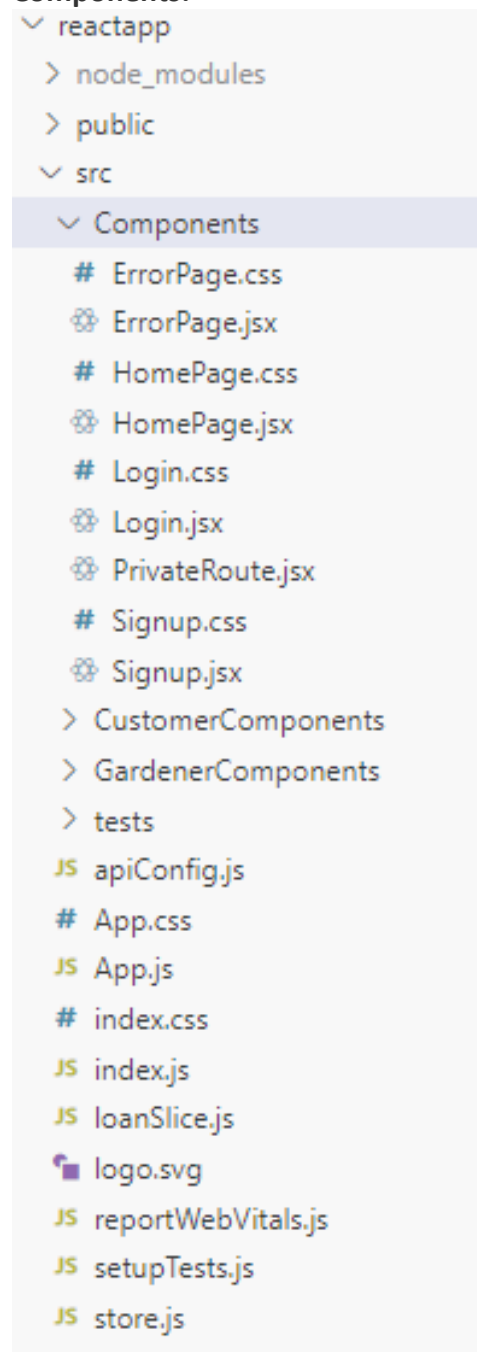
Ensure that the entire application uses the **DefaultAzureCredential** method for authentication.

Frontend Requirements:

- Create a folder named Components inside the src to store all common components. (Refer project structure screenshots).
- Create a folder named GardenerComponents inside the src to store all components for Gardener. (Refer project structure screenshots).
- Create a folder named CustomerComponents inside the src to store all components for Customer. (Refer project structure screenshots).
- You can create your own components based on the application requirements.

Project Folder Screenshot:

Components:



GardenerComponents:

▼ GardenerComponents

GardenerNavbar.css

⚙ GardenerNavbar.jsx

PlantForm.css

⚙ PlantForm.jsx

ViewPlant.css

⚙ ViewPlant.jsx

JS apiConfig.js

App.css

JS App.js

index.css

JS index.js

JS loanSlice.js

🖼 logo.svg

JS reportWebVitals.js

JS setupTests.js

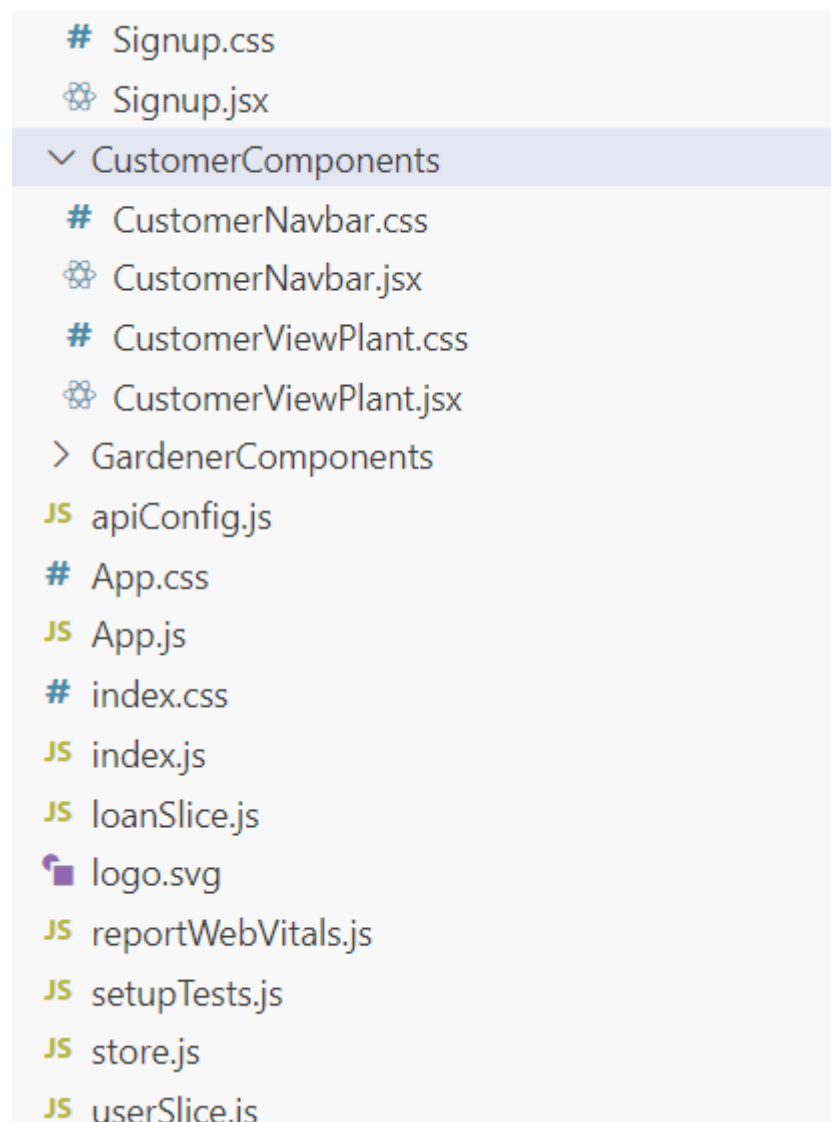
JS store.js

JS userSlice.js

📄 .babelrc

JS eslintrc.js

CustomerComponents:



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.

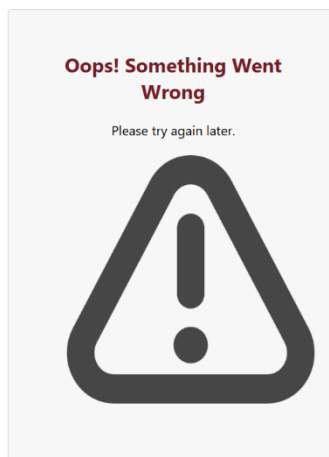
Exception Handling:

- 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 must be displayed if something goes wrong.



Note: Refer to the screenshot for additional details and design the forms accordingly.

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:

(GardenerNavbar component)

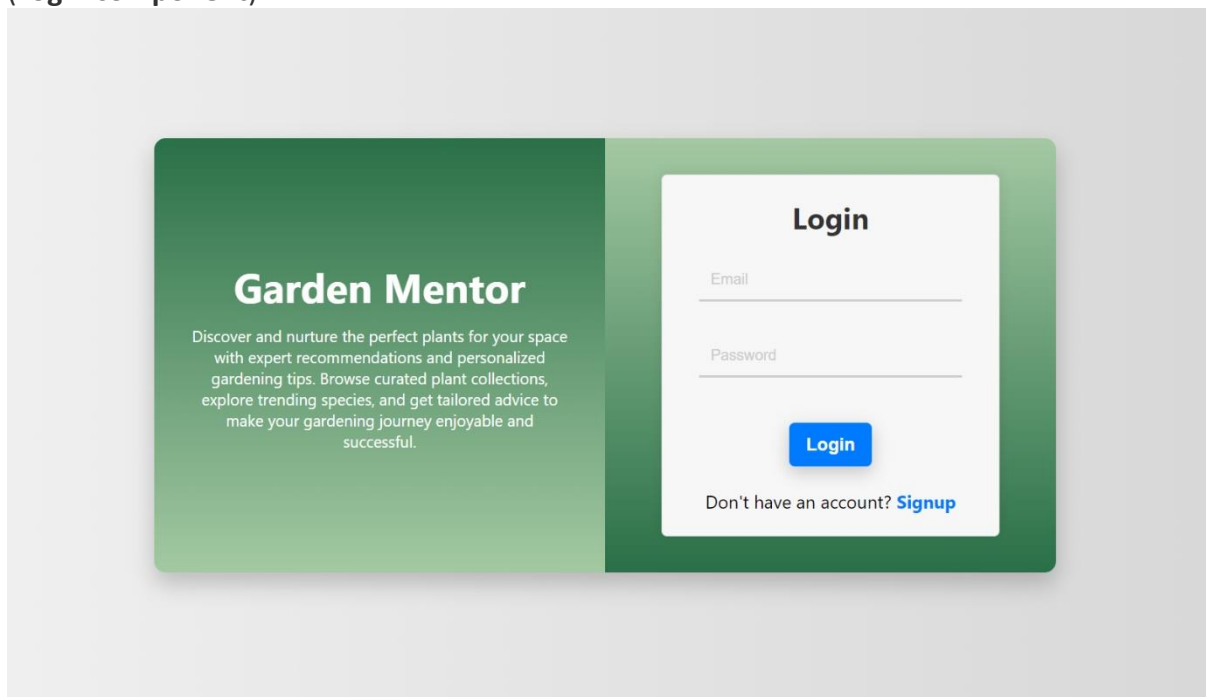


(CustomerNavbar component)



Landing Page:

(Login component)



Component features a heading "**Garden Mentor**" accompanied by an introductory message that provides an overview of the application with login form.

Common Screens (Gardener and Customer)

Signup Page:

(Signup component)

Clicking "**Signup**" in the login page will navigate to the registration page for both roles. Please refer to the screenshots below for validations.

Signup

User Name *

Username

Email *

Email

Mobile Number*

Mobile Number

Password *

Password

Confirm Password *

Confirm Password

Role*

Select Role

Submit

Already have an Account? [Login](#)

On clicking “Submit” button with empty field will display the validation message.

Signup

User Name *

Username

User Name is required

Email *

Email

Email is required

Mobile Number*

Mobile Number

Mobile number is required

Password *

Password

Password is required

Confirm Password *

Confirm Password

Confirm Password is required

Role*

Select Role

Please select a role

Submit

Already have an Account? [Login](#)

Regular expressions (Regex) are also included to validate email formats, mobile numbers, and to ensure password matching.

Signup

User Name *

DemoGardener

Email *

demogardener

Please enter a valid email

Mobile Number*

123123

Mobile number must be 10 digits

Password *

....

Password must be at least 6 characters

Confirm Password *

...|

Passwords do not match

Role*

Select Role

Please select a role

Submit

Already have an Account? [Login](#)

Signup

User Name *

DemoGardener

Email *

demogardener@gmail.com

Mobile Number*

1231231231

Password *

.....

Confirm Password *

.....

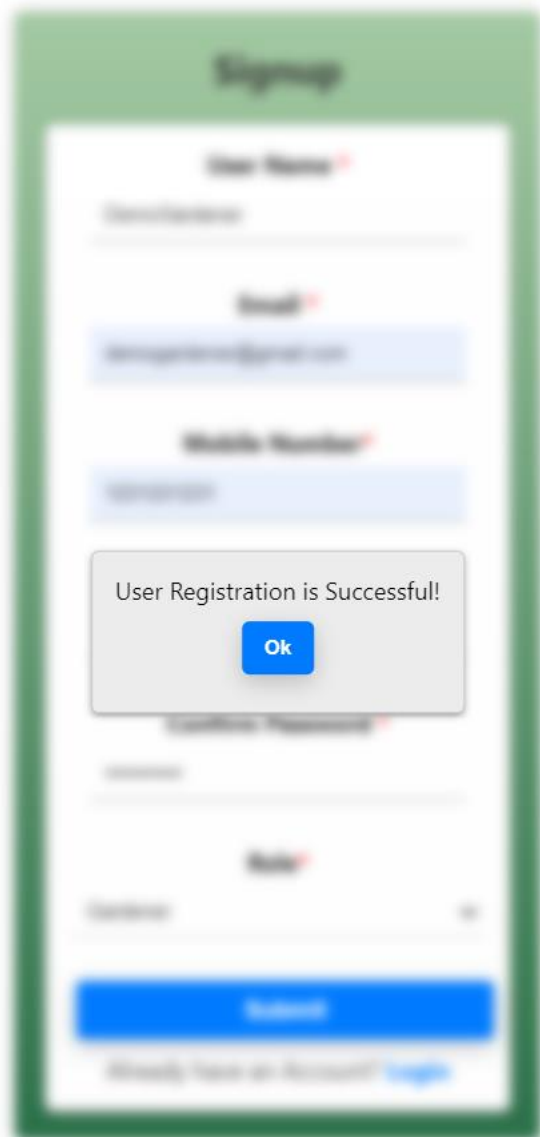
Role*

Gardener

Submit

Already have an Account? [Login](#)

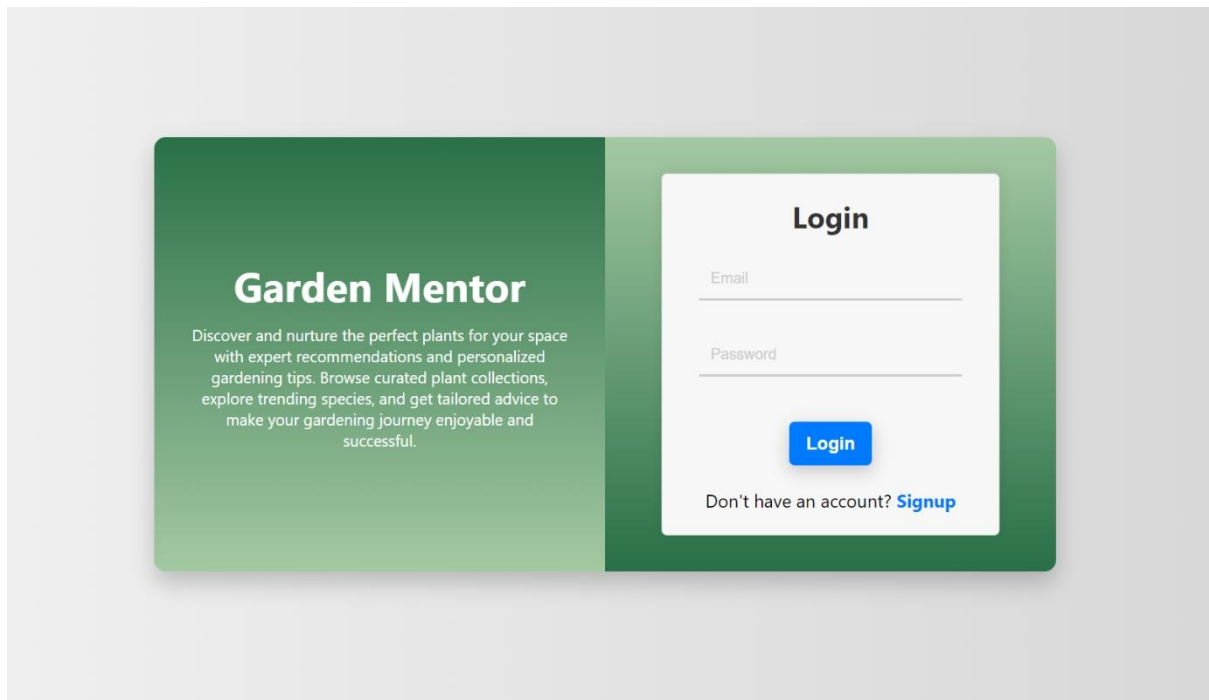
When the "Submit" button is clicked, upon successful submission, the user must be navigated to the login page.



Upon successful completion, a modal will appear, prompting users to click the "Ok" button. Upon clicking this button, users will be directed to the login page, signifying a successful signup process

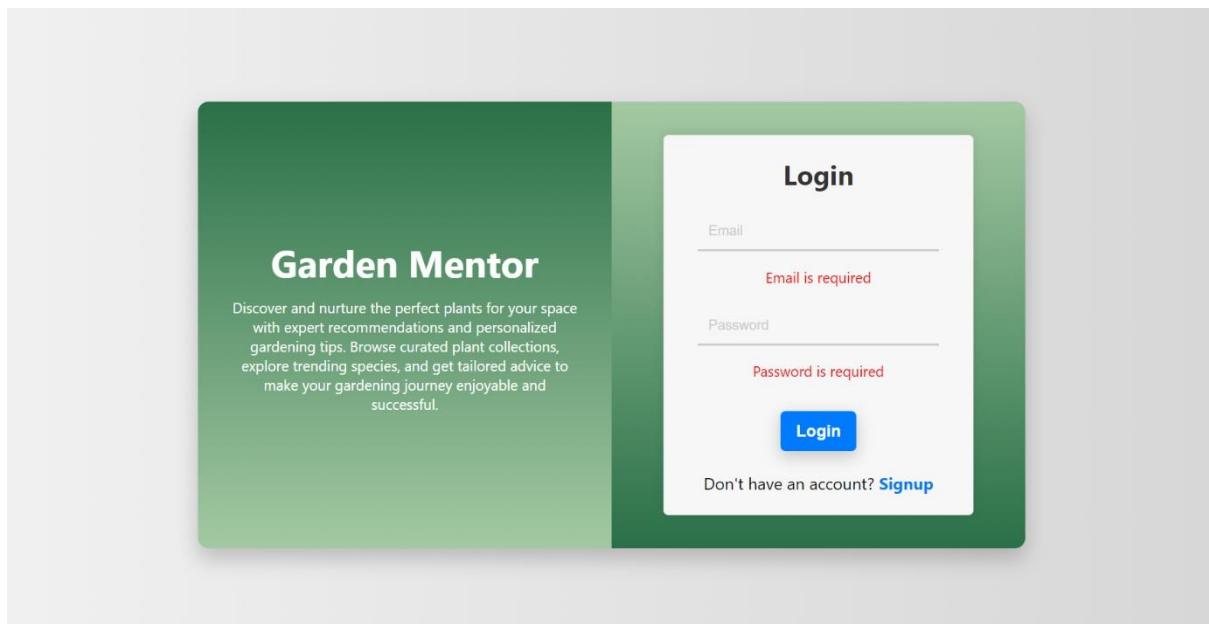
Login Page (login component)

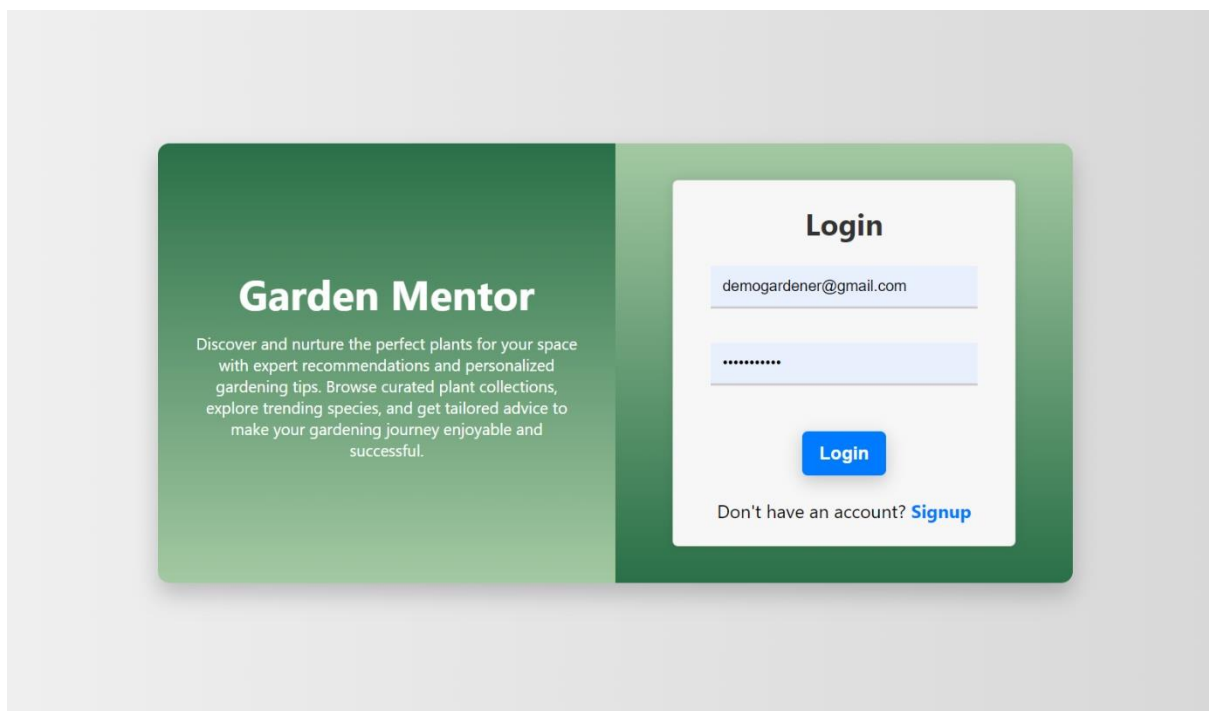
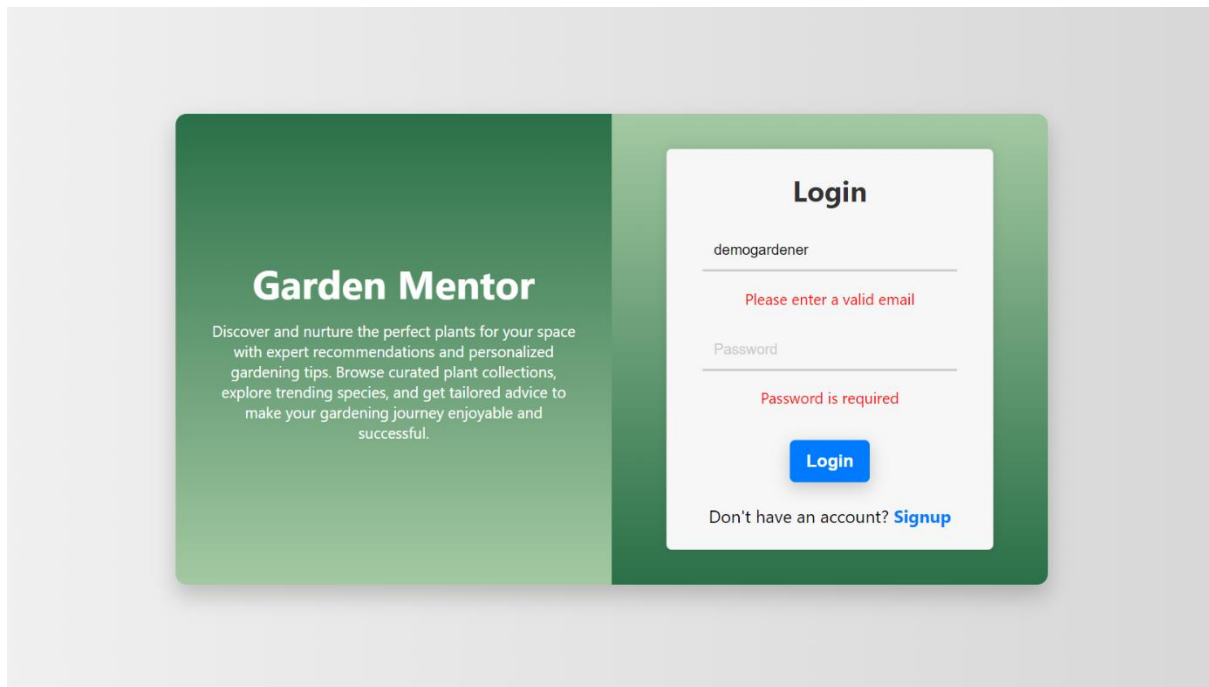
This page is used for logging in to the application. On providing the valid email and password, the user will be logged in.



Perform validations for email and password fields.

On Clicking "Login" button with empty field will display the validation message.





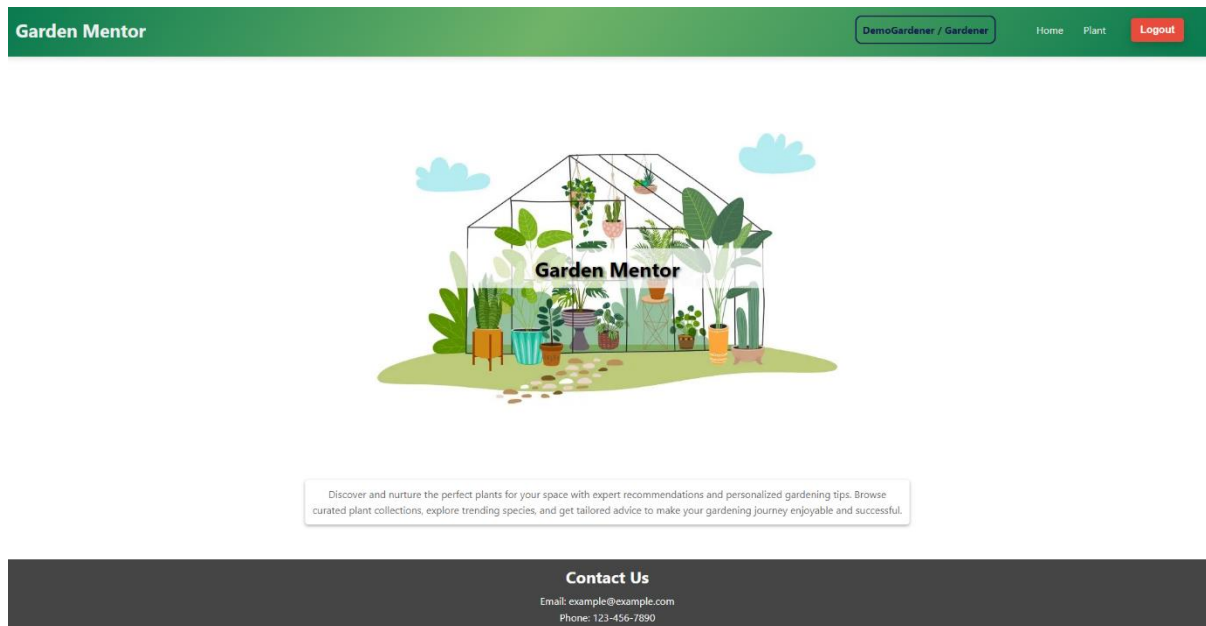
On Clicking the “Login” button, user will be navigated to the (GardenerNavbar or CustomerNavbar component) based on their roles.

Gardener Side:

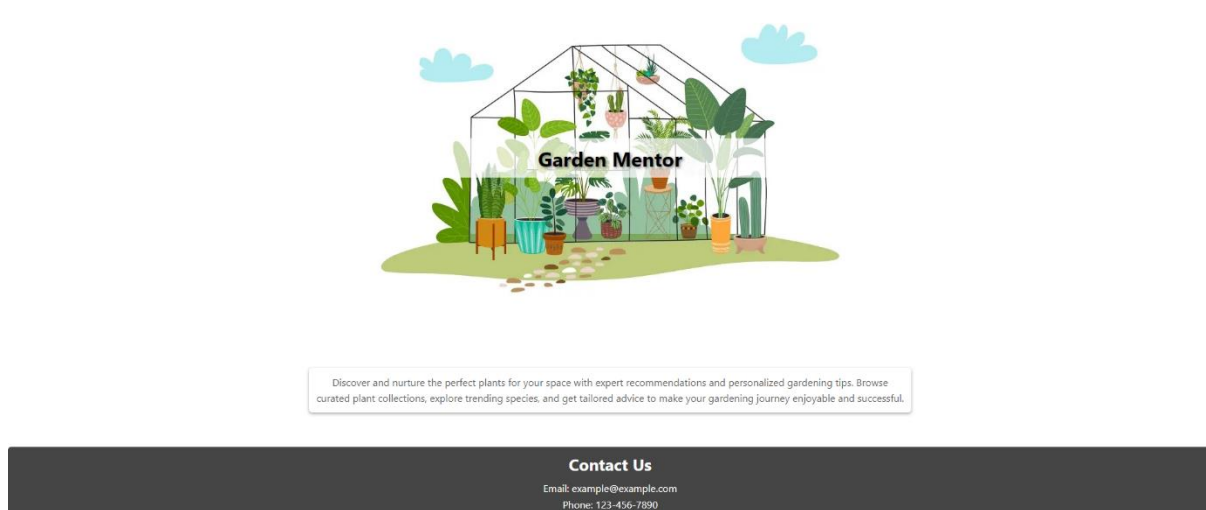
GardenerNavbar component:

Upon successful login, if the user is a Gardener, the (**GardenerNavbar** component) will be displayed. If the user is Customer, the (**CustomerNavbar** component) will be displayed.

Additionally, the role-based navigation bar will also display login information such as the username and role.

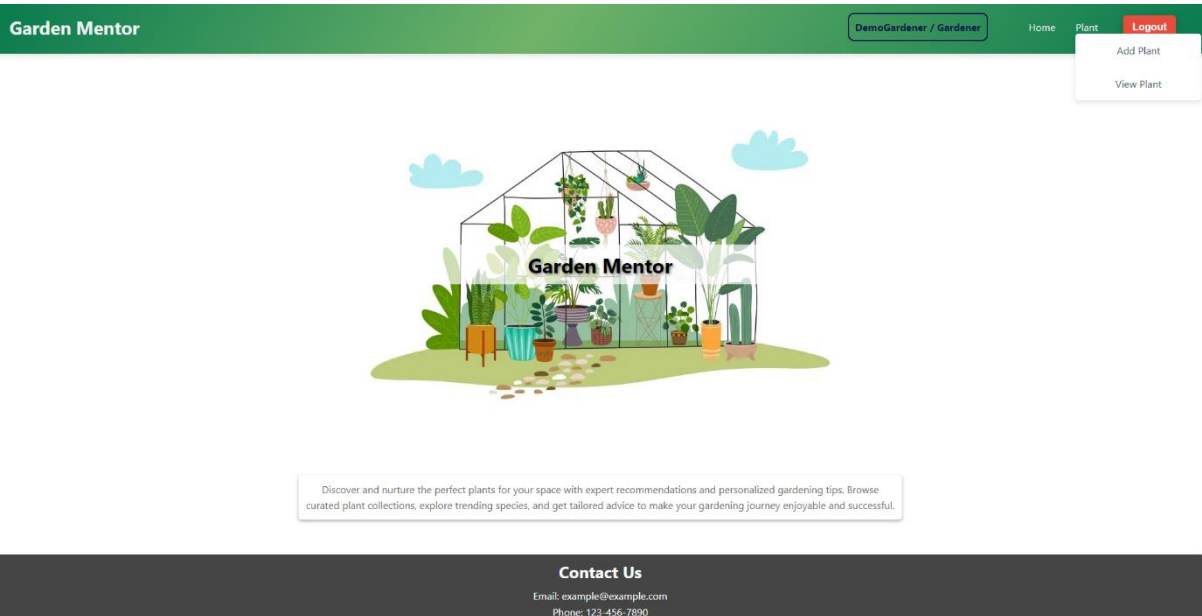


HomePage Component: This page is used to display the information about GardenMentor application. On clicking the 'Home' tab, user can view the information about the application.(gardenmentorcoverimage.jpg provided in public folder)



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

Clicking the "Plant" option in the navbar will reveal "Add Plant" and "View Plant" options, allowing the Gardener to add new Plants and view existing ones.



By clicking the “Add Plant” option will navigate to the “**PlantForm**” component, where the Gardener can add plant.

The screenshot displays the 'Create New Plant' form within the 'Garden Mentor' application. The form is centered on a white background and includes a 'Back' button in the top left corner. The form fields are: 'Name*' (text input), 'Category*' (dropdown menu with 'Select a category' as the placeholder), 'Price*' (text input), 'Tips*' (text area), and 'Plant Image*' (file upload button labeled 'Choose file' with 'No file chosen' as feedback). A blue 'Add Plant' button is located at the bottom of the form. The green navigation bar at the top of the page is identical to the one in the first screenshot.

On Clicking “Add Plant” button with empty field will display validations error message.

The screenshot shows the 'Create New Plant' form in the Garden Mentor application. The form is titled 'Create New Plant' and includes a 'Back' button. It contains several input fields: 'Name*' (text), 'Category*' (dropdown), 'Price*' (text), 'Tips*' (text area), and 'Plant Image*' (file upload). Each field has a red validation error message below it: 'Name is required', 'Category is required', 'Price is required', 'Tips are required', and 'Image is required'. At the bottom is a blue 'Add Plant' button. The top navigation bar is green with 'Garden Mentor' on the left and 'Home', 'Plant', and 'Logout' on the right.

On the 'Create New Plant' page, click the 'Category' option to select a plant category from a predefined list. This helps to organize and filter plant types more effectively.

Back

Create New Plant

Name*

Name

Category*

Select a category

Select a category

Indoor

Outdoor

Succulents

Ferns

Flowers

Herbs

Cacti

Plant Image*

Choose file

No file chosen

Add Plant

Back

Create New Plant

Name*

2026

Category*

Indoor

Price*

120


Tips*

To grow a healthy Jade plant, place it in bright, indirect sunlight for about 4-6 hours daily, water sparingly, allowing the soil to dry out completely between waterings, as Jade plants prefer drier conditions.

Plant Image*

Choose file

image.png



Add Plant

Upon successfully adding a Plant, a success modal with a "Close" button will be displayed.

Back

Create New Plant

Name*

Jade

Category*

Indoor

Price*

120

Tips*

Plant added successfully!


Close

Tips

To grow a healthy jade plant, place it in bright, indirect sunlight for about 4-6 hours daily. Water sparingly, allowing the soil to dry out completely between waterings, as jade plants prefer drier conditions.

Plant Image*

Choose file jadeplant.jpg



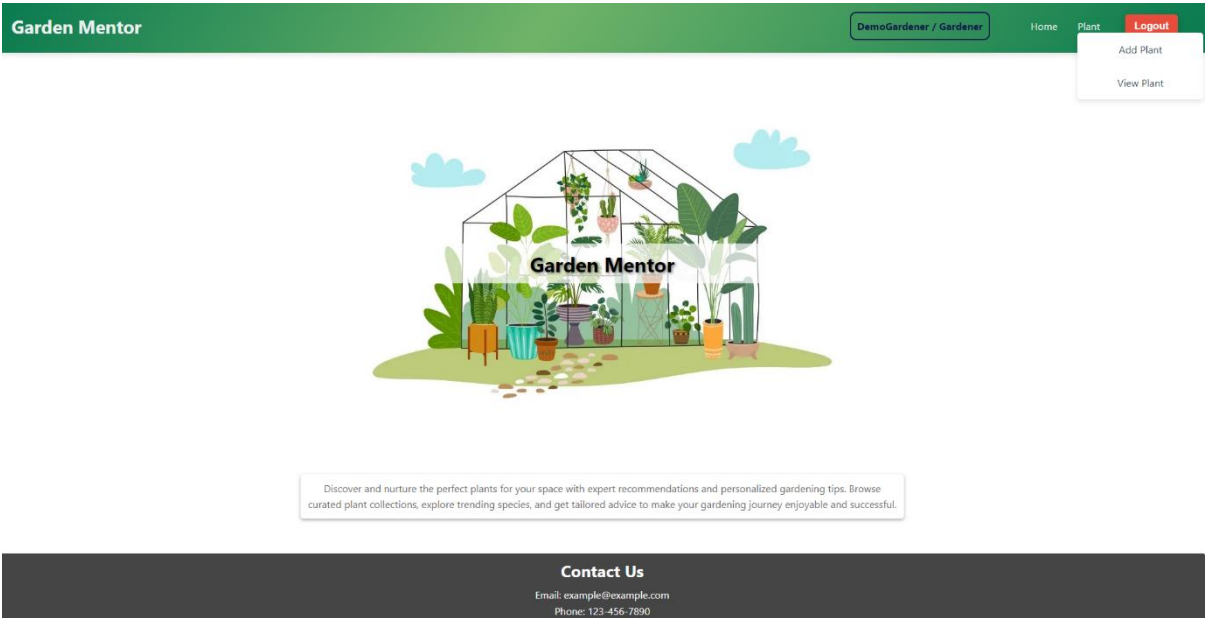
Add Plant

Clicking on “Close” button will navigate to “ViewPlant” component


Plants						
Image	Name	Category	Price	Tips	Action	
	Jade	Indoor	Rs. 120.00	To grow a healthy jade plant, place it in bright, indirect sunlight for about 4-6 hours daily. Water sparingly, allowing the soil to dry out completely between waterings, as jade plants prefer drier conditions.	Edit	Delete

ViewPlant Component:

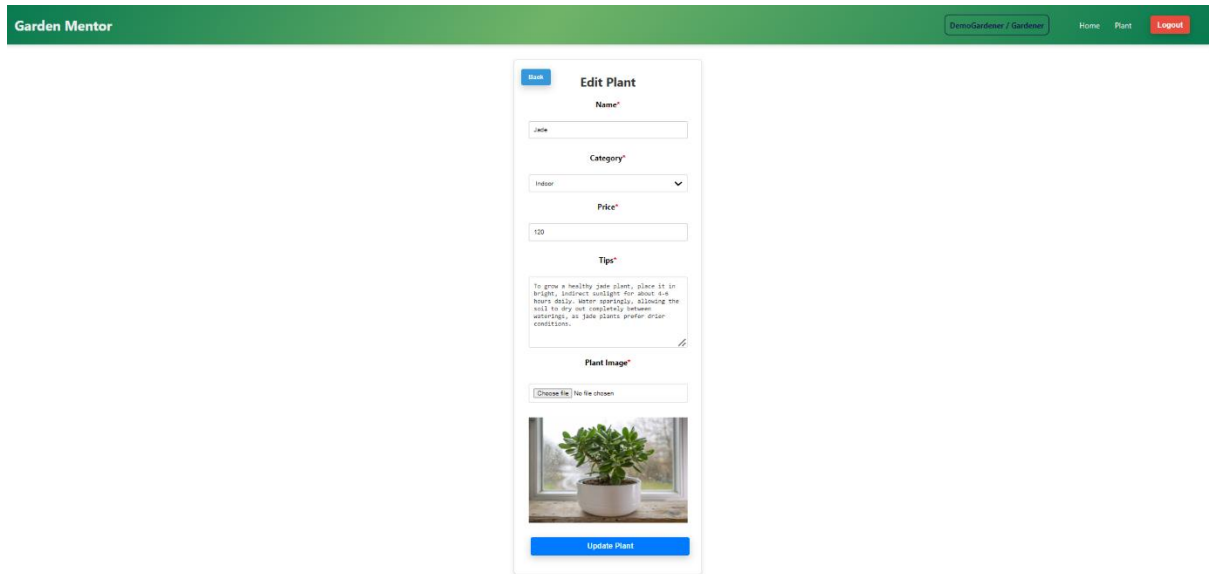
Clicking the “Plant” option in the navbar will reveal “Add Plant” and “View Plant” options, allowing the Gardener to add new Plants and view existing ones.



By clicking the “View Plant” option will navigate to the “**ViewPlant**” component, where the Gardener can view plant.

Plants					
Image	Name	Category	Price	Tips	Action
	Jade	Indoor	Rs. 120.00	To grow a healthy jade plant, place it in bright, indirect sunlight for about 4-6 hours daily. Water sparingly, allowing the soil to dry out completely between waterings, as jade plants prefer drier conditions.	<button>Edit</button> <button>Delete</button>

Clicking the "Edit" button will navigate to the **PlantForm** component, displaying the edit form prepopulated with the data of the selected plant.



The screenshot shows the 'Edit Plant' form in the Garden Mentor application. The form is prepopulated with the following data:

- Name:** Jade
- Category:** Indoor
- Price:** 120
- Tips:** To grow a healthy jade plant, place it in bright, indirect sunlight for about 4-6 hours daily. Water sparingly, allowing the soil to dry out completely between waterings, as jade plants prefer drier conditions.
- Plant Image:** A photograph of a jade plant in a white pot.

The form includes a 'Back' button at the top left and an 'Update Plant' button at the bottom right. A 'Choose file' button is also present next to the plant image.

Form includes “Back” button, will navigate to previous page.

Validation message is displayed on clicking “Update Plant” with empty field.

Back

Edit Plant

Name*

Name

Name is required

Category*

Indoor

▼

Price*

Price

Price is required

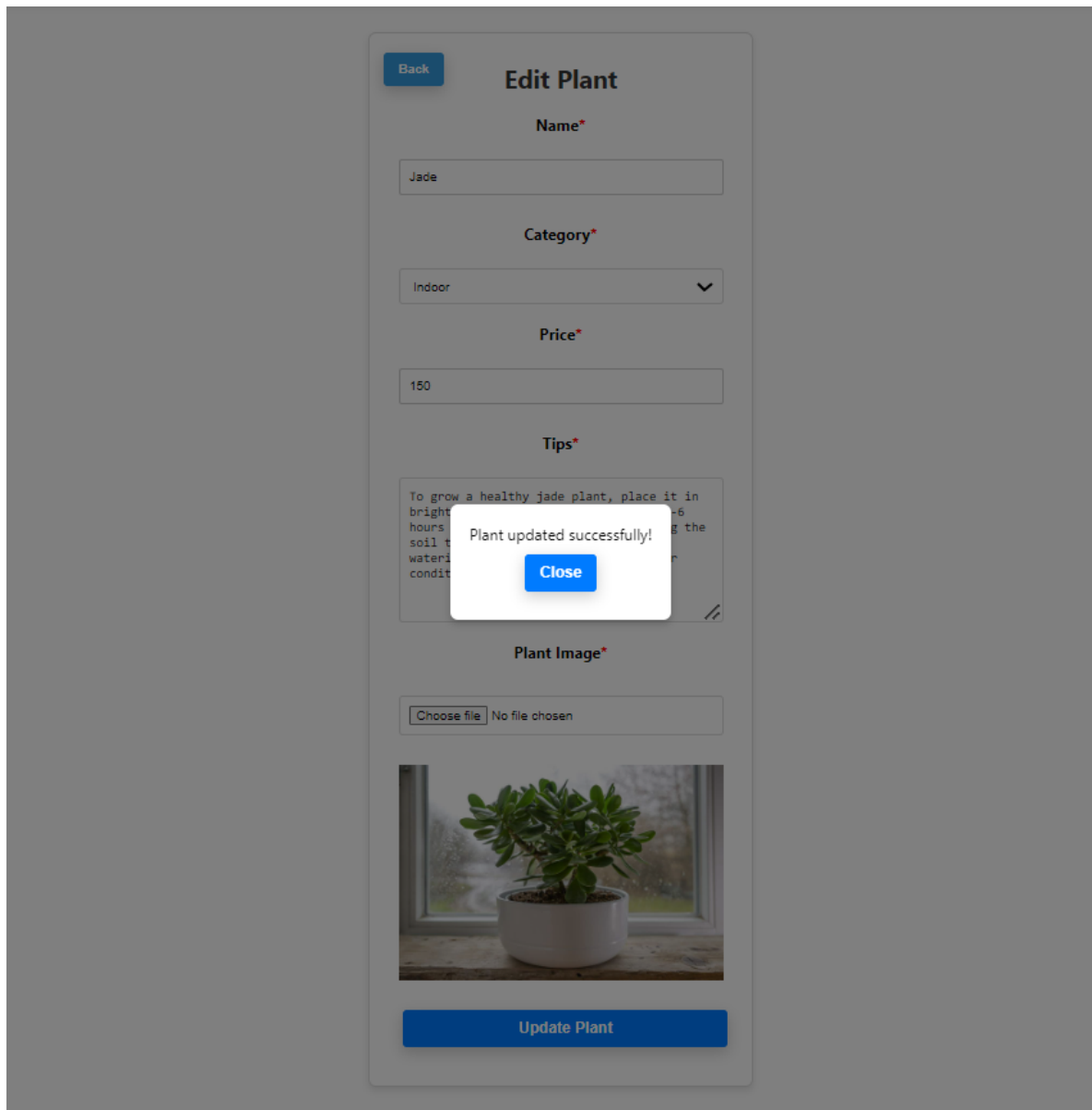
Tips*

Tips

Tips are required

Plant Image*

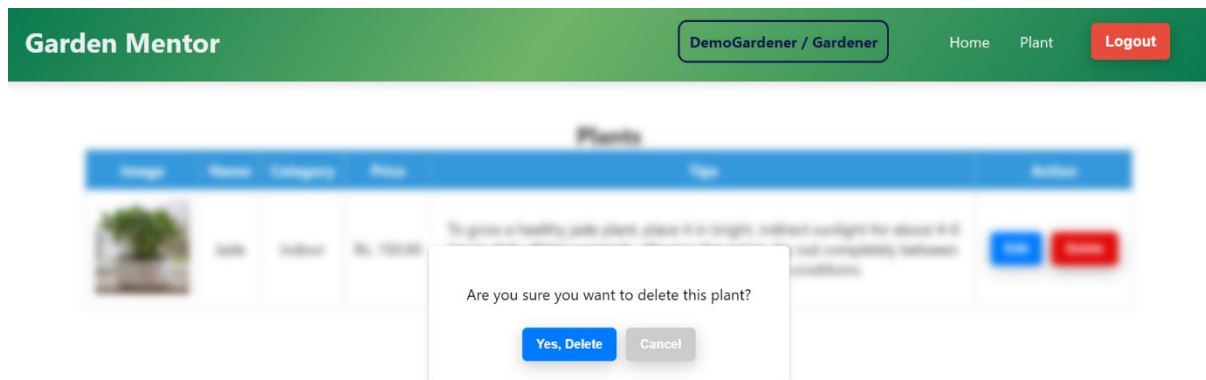
Upon successfully updating a Plant, a success modal with a "Close" button will be displayed.



Clicking on “Close” button will navigate to “**ViewPlant**” component

Plants					
Image	Name	Category	Price	Tips	Action
	Jade	Indoor	Rs. 120.00	To grow a healthy jade plant, place it in bright, indirect sunlight for about 4-6 hours daily. Water sparingly, allowing the soil to dry out completely between waterings, as jade plants prefer drier conditions.	Edit Delete

When clicking the "Delete" button, a confirmation modal will be displayed with the message **"Are you sure you want to delete this plant?"**



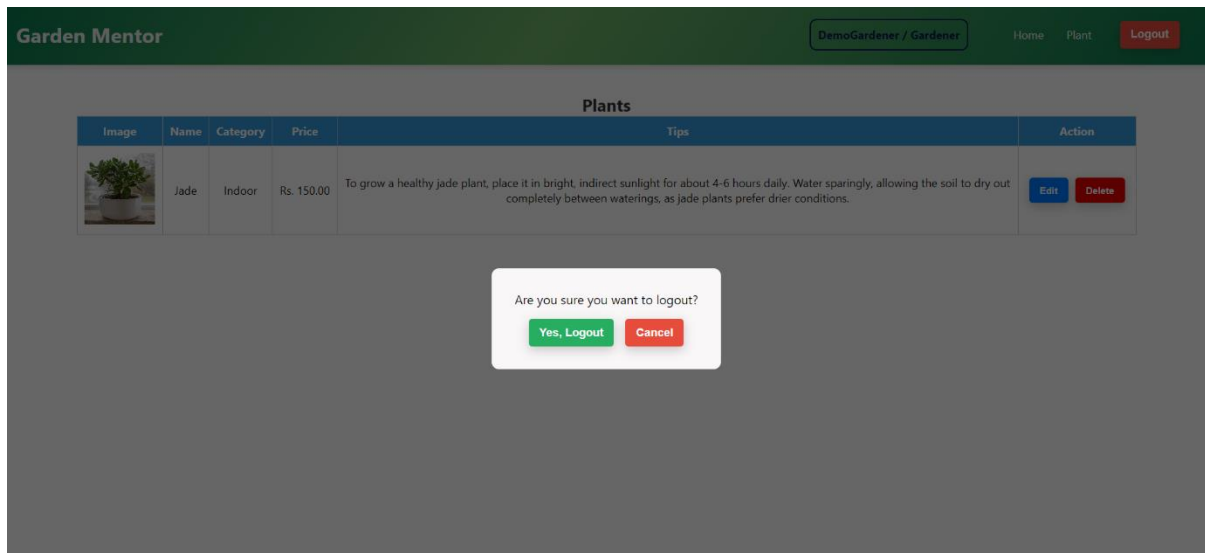
Clicking "Yes, Delete" will remove the selected Plant and refresh the displayed data.

If data is not available, then "Oops! No plants found." should be displayed.



Clicking "Cancel" will close the popup window.

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

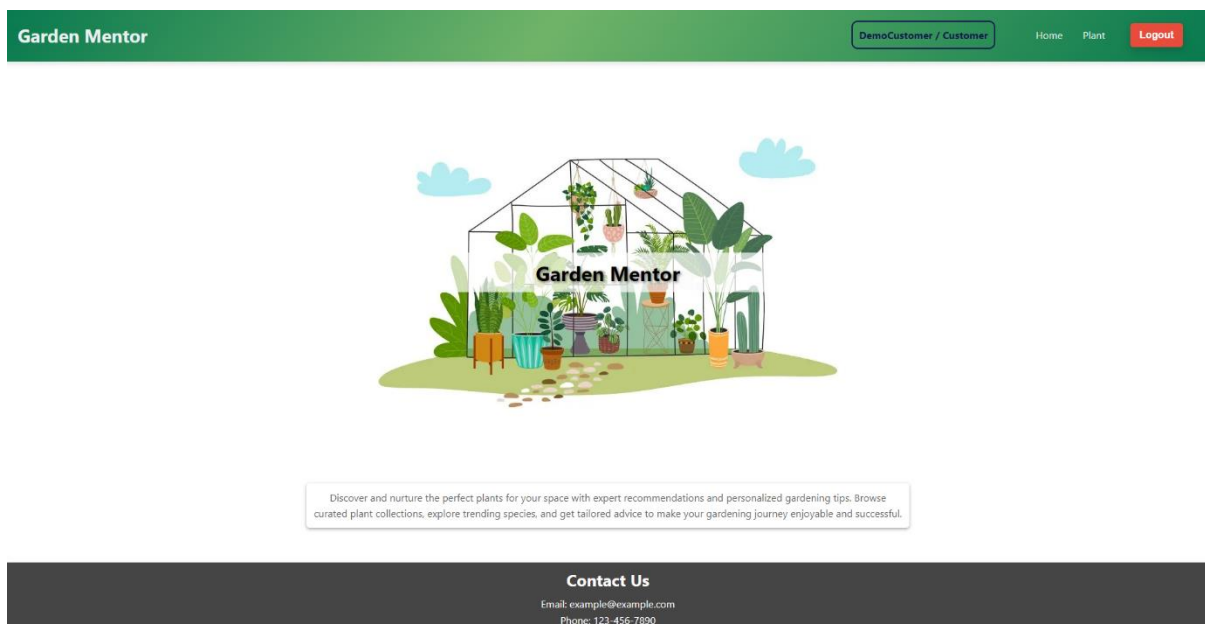


OnClicking "Yes, Logout" will navigate to the login component.
 OnClicking "Cancel" will close the modal window displayed.

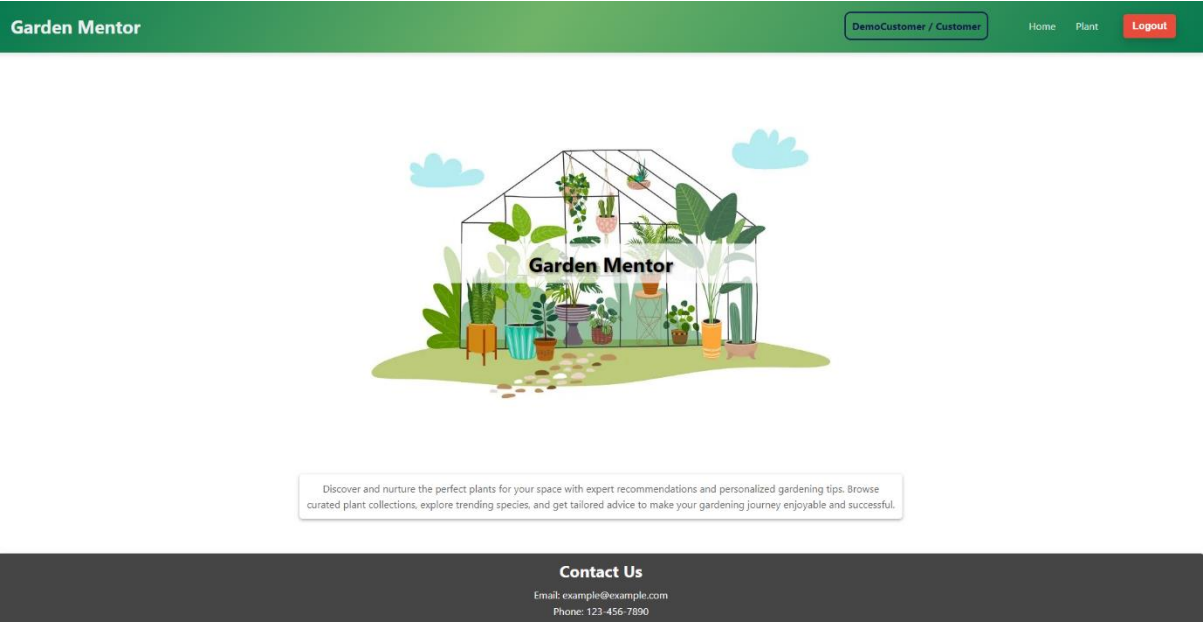
Customer Side:

CustomerNavbar component:

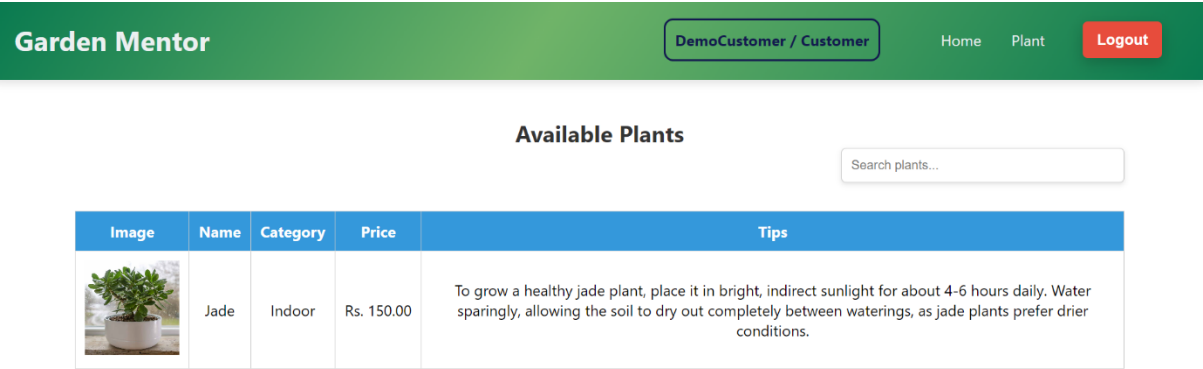
Upon successful login, if the user is an Gardener, the (**GardenerNavbar** component) will be displayed. If the user is Customer, the (**CustomerNavbar** component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.



HomePage Component: This page is used to display the information about GardenMentor application. On clicking the 'Home' tab, user can view the information about the application. (gardenmentorcoverimage.jfif provided in public folder)



On Clicking over the "Plant" item in the navbar, will navigate to "CustomerViewPlant" Component



Additionally with search functionality based on plant name .

Garden Mentor

demoadmin / Gardener

HomePlantLogout

Available Plants

tes

Image	Name	Category	Price	Tips
	test	Outdoor	Rs. 123.00	sdfs

If data is not available, then "Oops! No plants found." should be displayed.

Garden Mentor

DemoCustomer / Customer

HomePlantLogout

Available Plants

Search plants...

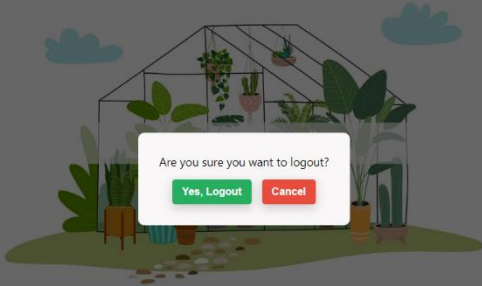
Image	Name	Category	Price	Tips
Oops! No plants found.				

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

Garden Mentor

democustomer / Customer

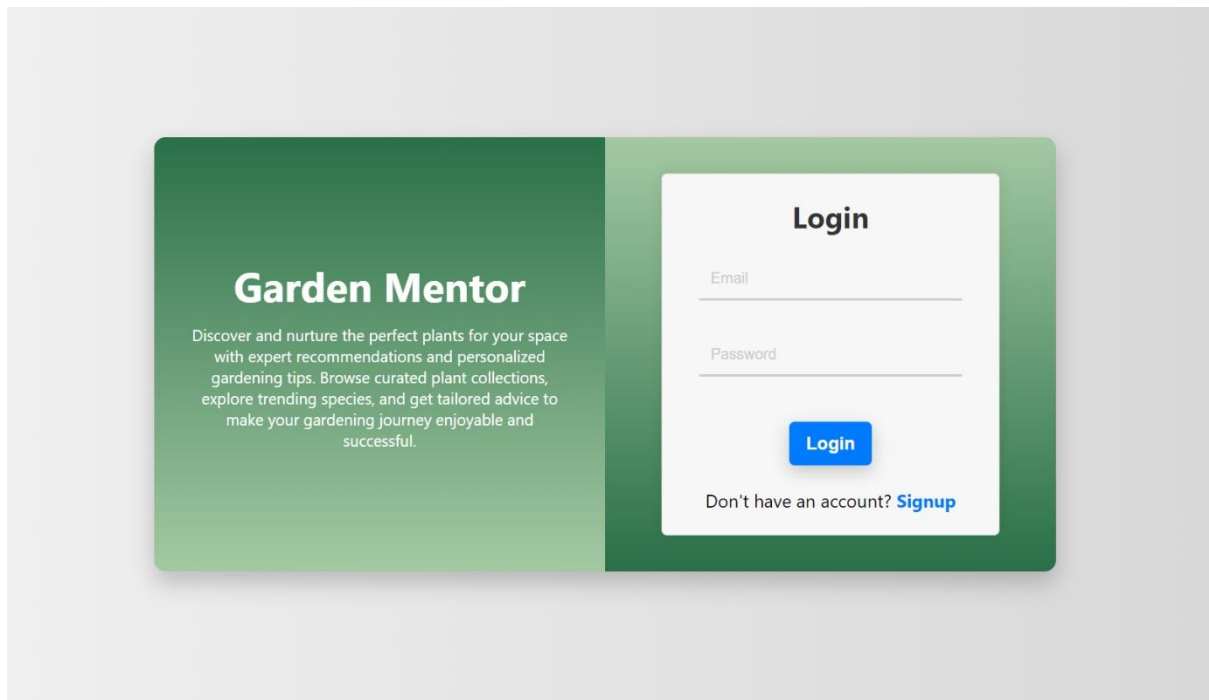
HomePlantLogout



Discover and nurture the perfect plants for your space with expert recommendations and personalized gardening tips. Browse curated plant collections, explore trending species, and get tailored advice to make your gardening journey enjoyable and successful.

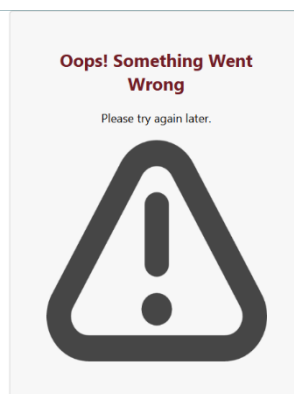
OnClicking "Cancel" will close the modal window displayed.

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



ErrorPage Component:

This page displays with “Oops! Something Went Wrong” message with the image (alert.png) provided in public folder.



Commands to Run the Project:

cd dotnetapp

Select the dotnet project folder

dotnet restore

This command will restore all the required packages to run the application.

dotnet run

To run the application in port 8080 (The settings preloaded click 8080 Port to View)

dotnet build

To build and check for errors

dotnet clean

If the same error persists clean the project and build again

To work with Entity Framework Core:

Install EF using the following commands:

dotnet new tool-manifest

dotnet tool install --local dotnet-ef --version 6.0.6

dotnet dotnet-ef --To check the EF installed or not

dotnet dotnet-ef migrations add "InitialSetup" --command to setup the initial creation of tables mentioned in DbContext

dotnet dotnet-ef database update --command to update the database

To Work with SQLServer:

(Open a New Terminal) type the below commands

sqlcmd -U sa

password: examlyMssql@123

1> create database appdb

2> go

>use appdb

>go

1> create table TableName(id int identity(1,1),.....)

2> go

Note:

The database name should be appdb.

Use the below sample connection string to connect the MsSql Server

```
private string connectionString = "User ID=sa; password=examlyMssql@123;  
server=localhost; Database=appdb; trusted_connection=false; Persist Security Info=False;  
Encrypt=False";
```


Platform Instructions to run the React project:

Step 1:

Use "cd reactapp" command to go inside the reactapp folder

Install Node Modules - "npm install"

Step 2:

Write the code inside src folder

Create the necessary components and enter the code

Step 3:

Command to run the project "npm start"

Step 4:

To run the test cases click the "Run Test Case" button.

Note :

Click PORT 8081 to view the result / output.

If any error persists while running the app, delete the node modules and reinstall them.

Cloud Deployment:

1. Introduction

1.1 Scope

The project involves deploying two separate applications (backend and frontend) to Azure App Services, integrating with Azure Key Vault for secrets management, using Azure SQL Database for data storage, and Azure Storage Account for storing uploaded images. The deployment will also include the creation and configuration of a service principal for secure access and necessary firewall rules for the SQL database.

Cloud Deployment

2.1 System Architecture

The system is divided into two main components: Backend and Frontend. Each component uses various Azure services to handle specific tasks.

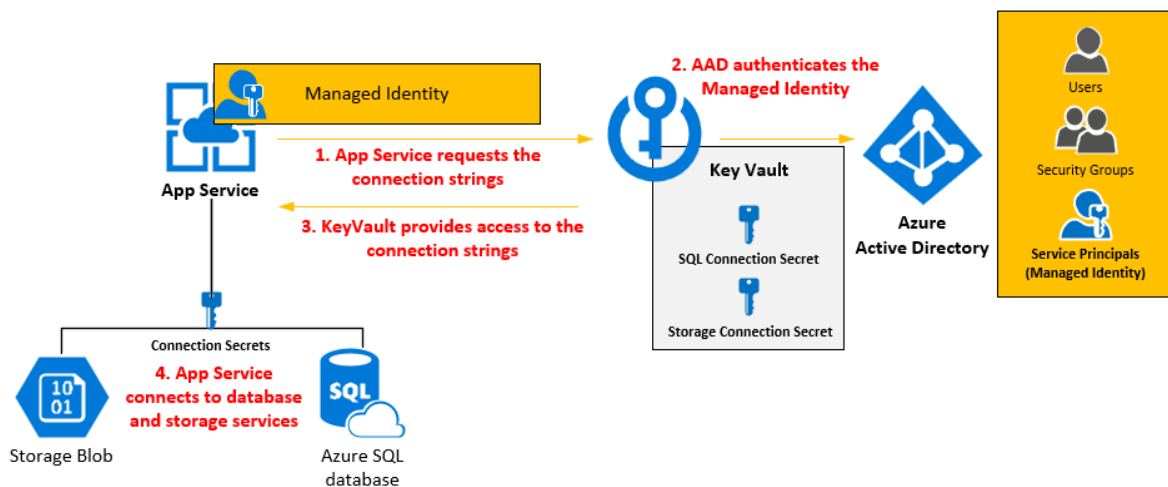
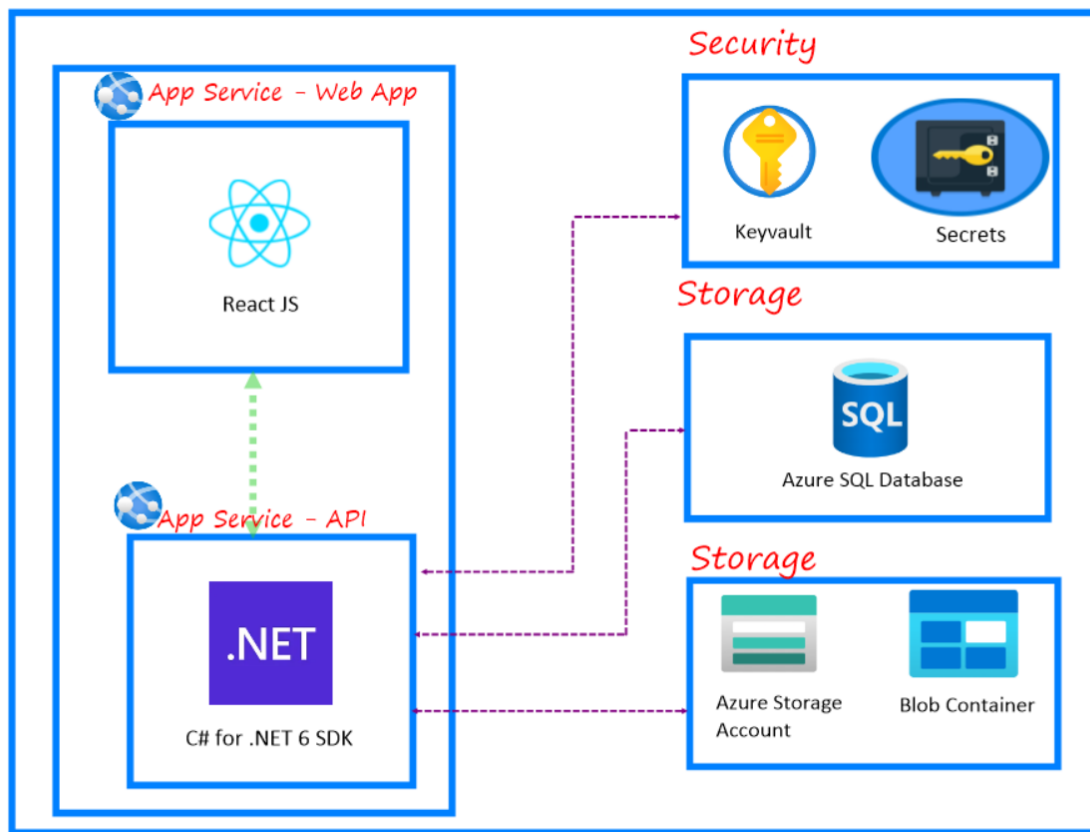
2.2 Backend Architecture

- **Azure App Service for .NET Backend**
 - App Name: "<Resource Group Name>bapp"
 - Configuration:
 - Deploy .NET backend using GitHub Actions
 - Configure to retrieve secrets from Azure Key Vault
 - Connect to Azure SQL Database using connection string from Key Vault

- Store uploaded images in Azure Storage Account
- **Azure Key Vault**
 - Key Vault Name: <Resource Group Name>
 - Usage:
 - Store database connection strings, storage account keys, and other secrets
- **Azure SQL Database**
 - Database Name: “<Backend App Service Name>-database”
 - Configuration:
 - Apply firewall rules to allow required IP addresses
 - Store connection string in Azure Key Vault
- **Azure Storage Account**
 - Storage Account Name: <Resource Group Name>
 - Configuration:
 - Use blob storage to store uploaded images
 - Store storage account connection string in Azure Key Vault
- **Service Principal**
 - Usage:
 - Created for access management
 - Used in environment variables and GitHub secrets for deployments
- **2.3 Frontend Architecture**
- **Azure App Service for React Frontend**
 - App Name: “<Resource Group Name>fapp”
 - Configuration:
 - Deploy React frontend using GitHub Actions
 - Configure to retrieve secrets from Azure Key Vault

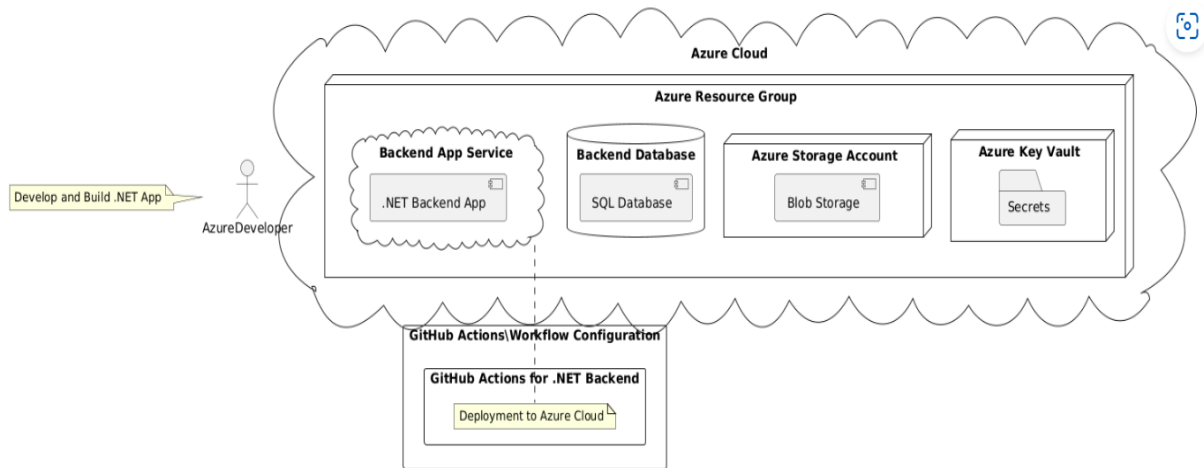
2.4 Architecture Diagram

[📦] Resource Group

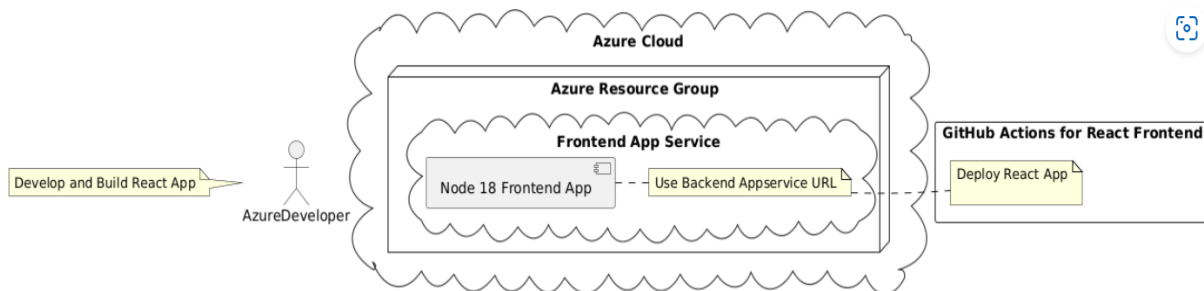


2.5 Flow Diagram

Backend Flow Diagram:



Frontend Flow Diagram:



3. Specific Requirements

3.1 Backend Services

Azure App Service for .NET Backend

- App Name: "<Resource Group Name>bapp"
- Deployment Steps:
 - Configure GitHub Actions workflow for deploying the .NET backend to bendapp24.
 - Run the GitHub Actions workflow.

Azure Key Vault

- Key Vault Name: <Resource Group Name>
- Configuration Steps:
 - Store necessary secrets (e.g., database connection strings, storage account keys) in edhubkvt24.
 - Update backend application to fetch secrets from Key Vault.

Azure SQL Database

- Database Name: “<Backend App Service Name>-database”
- Configuration Steps:
 - Create an Azure SQL Database.
 - Add firewall rules to allow required IP addresses.
 - Store the connection string in Azure Key Vault.
 - Update the backend application to use the connection string from Key Vault.

Azure Storage Account

- Storage Account Name: <Resource Group Name>
- Configuration Steps:
 - Create an Azure Storage Account with blob storage.
 - Store the storage account connection string in Azure Key Vault.
 - Update the backend application to use the storage account for storing uploaded images.

Service Principal

- Configuration Steps:
 - Create a service principal and assign necessary permissions.
 - Store the service principal credentials as environment variables and GitHub secrets.
 - Update GitHub Actions workflows to use the service principal for deployments.

3.2 Frontend Services

Azure App Service for React Frontend

- App Name: “<Resource Group Name>fapp”
- Deployment Steps:
 - Configure GitHub Actions workflow for deploying the React frontend to “<Resource Group Name>fapp”.
 - Run the GitHub Actions workflow.

3.3 GitHub Actions

GitHub Actions for .NET Backend

- Workflow Configuration:
 - Create a GitHub Actions workflow file to deploy the .NET backend to “<Resource Group Name>bapp”
 - Ensure the workflow includes steps for building the .NET application, logging into Azure, and deploying the application to Azure App Service.

GitHub Actions for React Frontend

- Workflow Configuration:

- Create a GitHub Actions workflow file to deploy the React frontend to frontendapp24.
- Ensure the workflow includes steps for building the React application, logging into Azure, and deploying the application to Azure App Service.

4.2 Integration with Azure Key Vault

- Steps:
 1. Store necessary secrets (e.g., database connection strings, storage account keys) in <Resource Group Name>
 2. Update both backend and frontend applications to fetch secrets from Key Vault.
 3. Deploy the applications.
- Expected Result: The applications should be able to retrieve and use secrets from the Key Vault successfully.

4.3 Azure SQL Database Configuration

- Steps:
 1. Create an Azure SQL Database.
 2. Add firewall rules to allow required IP addresses.
 3. Store the connection string in Azure Key Vault.
 4. Update the backend application to use the connection string from Key Vault.
 5. Deploy the backend application.
- Expected Result: The backend application should successfully connect to the SQL database using the connection string from Key Vault, and the SQL database should accept connections from the specified IP addresses.

4.4 Azure Storage Account Configuration

- Steps:
 1. Create an Azure Storage Account with blob storage.
 2. Store the storage account connection string in Azure Key Vault.
 3. Update the backend application to use the storage account for storing uploaded images.
 4. Deploy the backend application.
- Expected Result: The backend application should successfully store and retrieve images from the blob storage.

4.5 Service Principal Configuration

- Steps:
 1. Create a service principal and assign necessary permissions.
 2. Store the service principal credentials as environment variables and GitHub secrets.
 3. Update GitHub Actions workflows to use the service principal for deployments.

- Expected Result: The service principal should be successfully used for deployments and accessing resources, ensuring secure and seamless deployment.

General Instructions:

- You will be provided with an Azure Account Login Username and Password
- In the same account, a service principal's Client ID, Client Secret, and that Azure account's Subscription ID, Tenant ID will be provided.

Steps to be followed:

1. Create an Azure SQL database, a Storage Account, and a Keyvault
2. Complete the Backend Dotnet Development by mentioning the Azure SQL Database connection string and Azure Storage Account connection string along with the JWT Secret, JWT Valid issuer, JWT Valid Audience in the appsettings.json file. These connection strings and JWT Credentials totally 5 secrets must be stored in Azure Keyvault as secrets as mentioned below and it shouldn't be disclosed or called upon anywhere in the application development.
 - ASPNETCORE_ENVIRONMENT : Development
 - KeyVaultConfiguration__BlobSecret : blobsecret
 - KeyVaultConfiguration__ConnectionStringSecretName: sqlconnectionstring
 - KeyVaultConfiguration__JWTSecret : JWTSecret
 - KeyVaultConfiguration__JWTValidAudience : JWTValidAudience
 - KeyVaultConfiguration__URL : <https://<Your Keyvault Name>.vault.azure.net>
3. Once completed the Dotnet Backend Development, complete the Frontend React Development
4. Do the platform testing in 8080 and 8081 for backend and frontend respectively
5. Deploy two Azure App services for backend and frontend
6. In the Backend Dotnet application Azure App service, mention the Keyvault URL, all the 5 Keyvault secrets, also additional environmental variable 'ASPNETCORE' with the

value 'Development' totally 7 are to be given as Azure App settings Environmental Variables.

7. Complete the Backend Deployment to Azure App service via GitHub Actions which can use the Backend Publish Profile as Azure Web App Publish Profile can be used. So this should be stored in GitHub Secrets.
8. After the successful deployment of Dotnet backend application to respective Azure App service, build the Front end React application with the Azure Backend App service URL
9. Once completed the build process successfully, push the Frontend code with 'build' folder to GitHub.
10. Deploy the Frontend application with the already built 'build' folder to the second Azure App service which will be in reactapp build folder.
11. In the deployed Front end React Azure App service, mention in the App settings Environmental variable with the following Backend App Service URL:
 - API_BASE_URL: "<https://<Your Backend App Service Name>.azurewebsites.net>"
12. For the React application site access, give the following command in the Frontend App Service Configuration App Settings Start up command :

pm2 serve /home/site/wwwroot --no-daemon --spa