FarmConnect

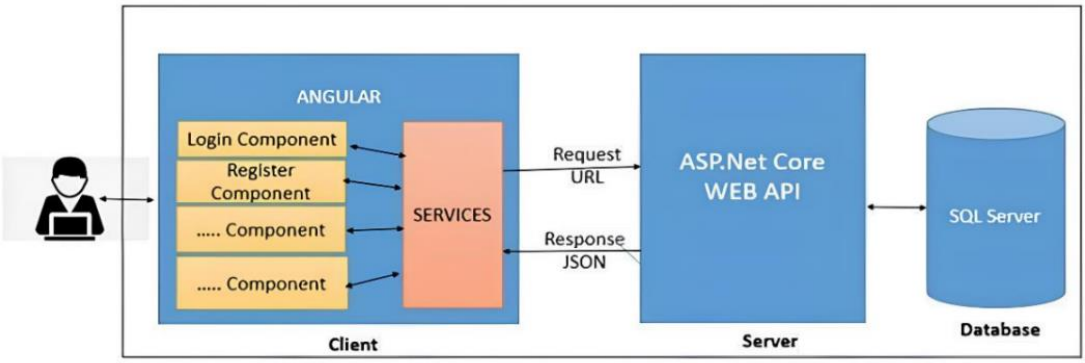
Overview:

FarmConnect simplifies livestock management for owners and suppliers. It's a user-friendly platform that streamlines tasks like tracking livestock health, managing medicine and feed inventory, and handling resource requests. With FarmConnect, users can efficiently care for livestock and ensure a steady supply of essential resources, fostering productivity and collaboration in the agricultural community.

Users of the System:

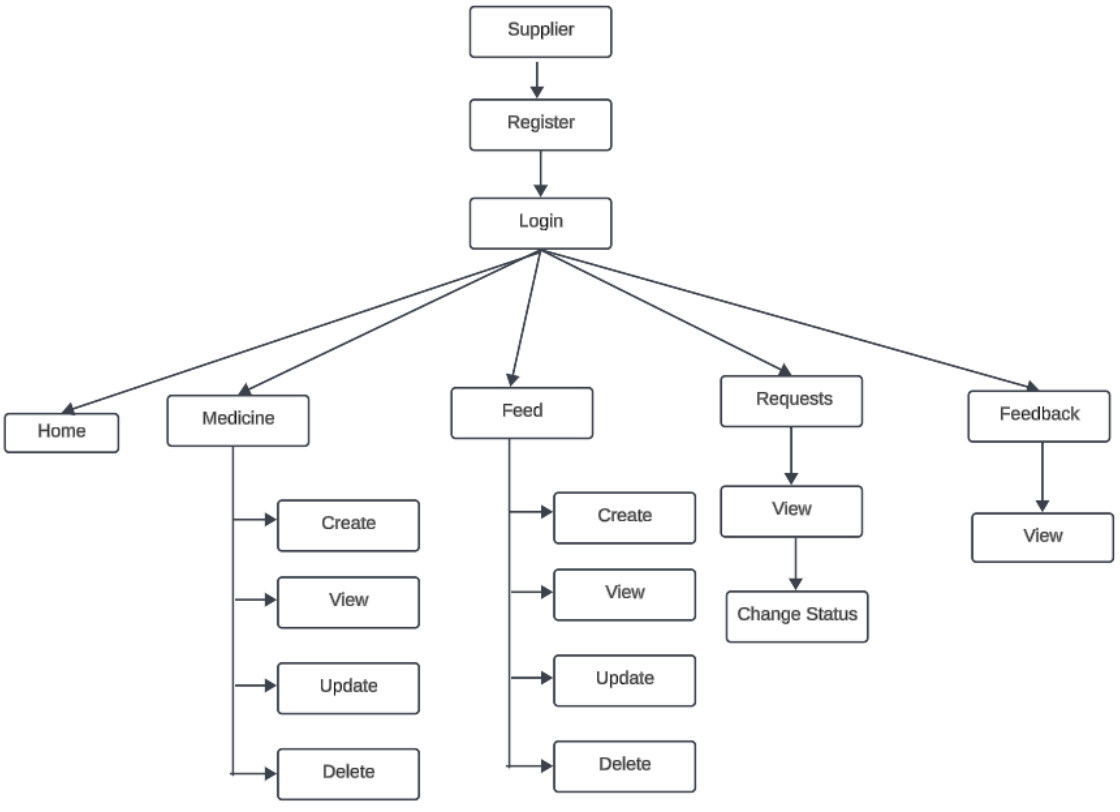
1. Supplier
2. Owner

System Architectural Diagram:

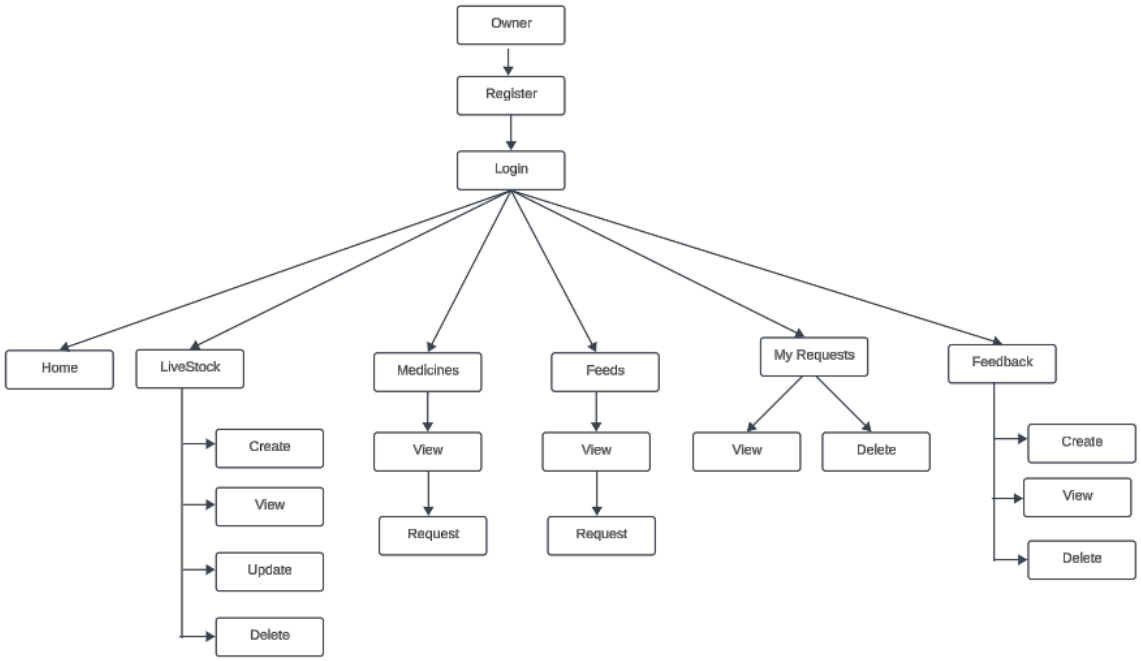


Supplier Actions:

Flow Diagram:



Owner Actions:
Flow Diagram:



Modules of the Application:
Owner:

1. Register
2. Login
3. LiveStock
4. Medicines
5. Feeds
6. My Requests
7. Feedback

User:

1. Register
2. Login
3. Home
4. Medicine
5. Feed
6. Requests
7. Feedback

Technology Stack
Front End
Angular 10+, HTML, CSS

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

Application assumptions:

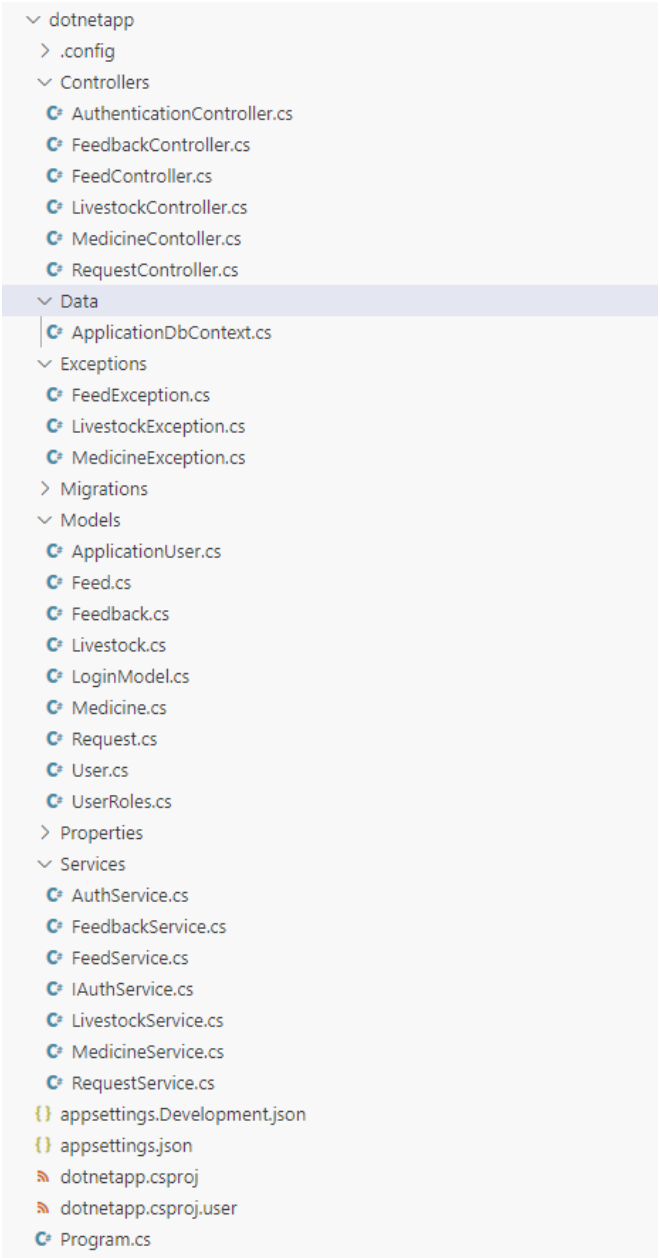
1. The login page should be the first page rendered when the application loads.
2. Manual routing should be restricted by implementing Auth Guard, utilizing the canActivate interface. For example, if the user enters as http://localhost:8080/dashboard or

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<Feedback> Feedbacks { get; set; }
public DbSet<User> Users { get; set; }
public DbSet<Feed> Feeds { get; set; }
public DbSet<Medicine> Medicines { get; set; }
public DbSet<Request> Requests { get; set; }
public DbSet<Livestock> Livestocks { 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 (Admin or User) and all user information.
Properties:
- UserId: int
- Email: string
- Password: string
- Username: string
- MobileNumber: string
- UserRole: string (**Owner/Supplier**)

Feed (Models / Feed.cs):
This class stores information about a feed.
Properties:
- FeedId: int
- FeedName: string
- Type: string
- Description: string
- Quantity: int
- Unit: string
- PricePerUnit: decimal
- Image: string
- UserId: int
- User?: User

Medicine (Models / Medicine.cs):
This class represents a medicine.
Properties:
- MedicineId: int
- MedicineName: string
- Brand: string
- Category: string
- Description: string
- Quantity: int
- Unit: string
- PricePerUnit: decimal
- Image: string
- UserId: int
- User?: User

Request (Models / Request.cs):
This class represents a request made by a user for feed or medicine related to livestock.
Properties:
- RequestId: int
- RequestType: string
- FeedId: int?
- Feed?: Feed
- MedicineId: int?
- Medicine?: Medicine
- UserId: int
- User?: User
- LivestockId: int
- Livestock?: Livestock
- Quantity: int
- Status: string

• RequestDate: DateTime

Livestock (Models / Livestock.cs):
This class represents information about livestock.
Properties:
• LivestockId: int
• Name: string
• Species: string
• Age: int
• Breed: string
• HealthCondition: string
• Location: string
• VaccinationStatus: string
• UserId: int
• User?: User

Feedback (Models / Feedback.cs):
This class represents feedback submitted by users.
Properties:
- FeedbackId: int
- UserId: int
- User?: User
- FeedbackText: string
- Date: 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. Admin: string - Represents the role of an admin user.
2. User: string - Represents the role of a regular user.

ApplicationUser (Models / ApplicationUser.cs):
This class represents a user in the application, inheriting from **IdentityUser** class.
Property:
• Name: string (Max length 30)

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

For example, you might throw a **MedicineException**:
1. When attempting to add a medicine with the same name and brand as an existing one.
2. When trying to update a medicine with the same name and brand as an existing one.
3. When attempting to delete a medicine that is referenced in requests.

Exceptions: (Exceptions / FeedException.cs)
Inside Exceptions folder, create the exception file named FeedException (FeedException.cs).

Purpose: The FeedException class provides a mechanism for handling exceptions related to feed operations within the application.
Namespace: It should be located within the dotnetapp.Exceptions namespace.
Inheritance: Inherits from the base Exception class, enabling it to leverage existing exception handling mechanisms.
Constructor: Contains a constructor that accepts a message parameter, allowing you to specify custom error messages when throwing exceptions.

For example, you might throw a **FeedException**:

1. When attempting to add a feed with the same name as an existing one.
2. When trying to update a feed with the same name as an existing one.
3. When attempting to delete a feed that is referenced in requests.

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.

Constructor:

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

FeedService (Services / FeedService.cs)
This service class provides methods to interact with feed data stored in the database.

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

Functions:

1. public async Task<IEnumerable<Feed>> GetAllFeeds()
   a. Retrieves and returns all feeds from the database.

2. public async Task<Feed> GetFeedById(int feedId):
   a. Retrieves a feed from the database with the specified **feedId**.

3. public async Task<IEnumerable<Feed>> GetFeedsByUserId(int userId)
   a. Retrieves and returns all feeds from the database with the specified **userId**.

4. public async Task<bool> AddFeed(Feed feed)
   a. Checks if a feed with the same name already exists in the database.
   b. If a feed with the same name and type exists, throws a **FeedException** with the message "**Feed with the same name and type already exists**".
   c. If no feed with the same name and type exists, adds the new feed to the database.
   d. Saves changes asynchronously to the database.
   e. Returns **true** upon successful insertion of the new feed.

5. public async Task<bool> UpdateFeed(int feedId, Feed feed)
   a. Retrieves the existing feed from the database based on the provided feedId.
   b. If no feed with the specified feedId is found, returns false.
   c. Checks if a feed with the same name already exists in the database, excluding the current feed being updated.
   d. If a feed with the same name exists, throws a **FeedException** with the message "**Feed with the same name and type already exists**".

e. Updates the existing feed with the values from the provided feed object.
f. Saves changes asynchronously to the database.
g. Returns **true** upon successful update of the feed.

6. public async Task<bool> DeleteFeed(int feedId)

a. Retrieves the feed from the database based on the provided feedId.
b. If no feed with the specified feedId is found, returns false.
c. Checks if the feed is referenced in any requests.
d. If the feed is referenced, throws a **FeedException** with the message "**Feed cannot be deleted, it is referenced in requests**".
e. Removes the feed from the database.
d. Saves changes asynchronously to the database.
f. Returns true upon successful deletion of the feed.

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

Constructor:

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

Functions:

1. public async Task<IEnumerable<Medicine>> GetAllMedicines()
a. Retrieves and returns all medicines from the database.

2. public async Task<Medicine> GetMedicineById(int medicineId)
a. Retrieves a medicine from the database with the specified medicineId.

3. public async Task<IEnumerable<Medicine>> GetMedicinesByUserId(int userId)
a. Retrieves and returns all medicines from the database with the specified userId.

4. public async Task<bool> AddMedicine(Medicine medicine)
a. Checks if a medicine with the same name and brand already exists in the database.
b. If a medicine with the same name and brand exists, throws a MedicineException with the message "Medicine with the same name and brand already exists".
c. If no medicine with the same name and brand exists, adds the new medicine to the database.
d. Saves changes asynchronously to the database.
e. Returns true upon successful insertion of the new medicine.

5. public async Task<bool> UpdateMedicine(int medicineId, Medicine medicine)
a. Retrieves the existing medicine from the database based on the provided medicineId.
b. If no medicine with the specified medicineId is found, returns false.
c. Checks if a medicine with the same name and brand already exists in the database, excluding the current medicine being updated.
d. If a medicine with the same name and brand exists, throws a **MedicineException** with the message "**Medicine with the same name and brand already exists**".
e. Updates the existing medicine with the values from the provided medicine object.
f. Saves changes asynchronously to the database.
g. Returns true upon successful update of the medicine.

6. public async Task<bool> DeleteMedicine(int medicineId)
a. Retrieves the medicine from the database based on the provided medicineId.
b. If no medicine with the specified medicineId is found, returns false.
c. Checks if the medicine is referenced in any requests.
d. If the medicine is referenced, throws a MedicineException with the message "**Medicine cannot be deleted, it is referenced in requests**".
e. Removes the medicine from the database.
f. Saves changes asynchronously to the database.
g. Returns true upon successful deletion of the medicine.

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

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

Functions:

1. public async Task<Livestock> GetLivestockById(int livestockId)
a. Retrieves a livestock record from the database with the specified livestockId.

2. public async Task<IEnumerable<Livestock>> GetLivestocksByUserId(int userId)
a. Retrieves and returns all livestock records from the database associated with the specified userId.

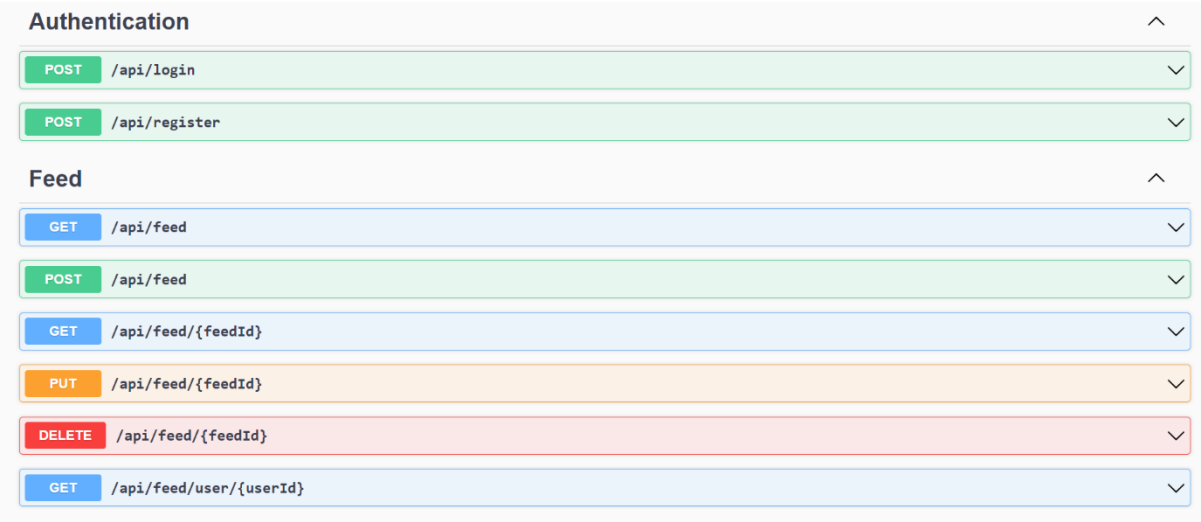
3. public async Task<bool> AddLivestock(Livestock livestock)
a. Checks if a livestock with the same name, breed, and species already exists in the database.
b. If a livestock with the same name, breed, and species exists, throws a LivestockException with the message "**Livestock with the same name, breed, and species already exists**".
c. If no livestock with the same name, breed, and species exists, adds the new livestock to the database.
d. Saves changes asynchronously to the database.
e. Returns true upon successful insertion of the new livestock.

4. public async Task<bool> UpdateLivestock(int livestockId, Livestock livestock)
a. Retrieves the existing livestock from the database based on the provided **livestockId**.
b. If no livestock with the specified livestockId is found, returns false.
c. Checks if a livestock with the same name, breed, and species already exists in the database, excluding the current livestock being updated.
d. If a livestock with the same name, breed, and species exists, throws a **LivestockException** with the message "**Livestock with the same name, breed, and species already exists**".
e. Updates the existing livestock with the values from the provided livestock object.
f. Saves changes asynchronously to the database.
g. Returns true upon successful update of the livestock.

5. public async Task<bool> DeleteLivestock(int livestockId)
a. Retrieves the livestock from the database based on the provided livestockId.
b. If no livestock with the specified livestockId is found, returns false.
c. Checks if the livestock is referenced in any requests.
d. If the livestock is referenced, throws a LivestockException with the message "**Livestock cannot be deleted, it is referenced in requests**".
e. Removes the livestock from the database.
f. Saves changes asynchronously to the database.
g. Returns true upon successful deletion of the livestock.

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

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



Functions:

1. public async Task<Request> GetRequestById(int requestId)
a. Retrieves a request from the database with the specified requestId.

2. public async Task<bool> AddRequest(Request request)
a. Adds the new request to the database.
b. Saves changes asynchronously to the database.
c. Returns true upon successful insertion of the new request.

3. public async Task<bool> UpdateRequest(int requestId, Request request)
a. Retrieves the existing request from the database based on the provided requestId.
b. If no request with the specified requestId is found, returns false.
c. Updates the existing request with the values from the provided request object.
d. Saves changes asynchronously to the database.
e. Returns true upon successful update of the request.

4. public async Task<IEnumerable<Request>> GetRequestsByUserId(int userId)
a. Retrieves and returns all requests associated with a specific userId from the database.

5. public async Task<IEnumerable<Request>> GetRequestsByUserIdInMedicineOrFeed(int userId)
a. Retrieves and returns all requests associated with a specific userId where the request is related to either Medicine or Feed.

6. public async Task<bool> DeleteRequest(int requestId)
a. Retrieves the request from the database based on the provided **requestId**.
b. If no request with the specified **requestId** is found, returns false.
c. Removes the request from the database.
d. Saves changes asynchronously to the database.
e. Returns true upon successful deletion of the request.

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():**
a. Retrieves all feedbacks from the database.

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

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

4. public async Task<bool> DeleteFeedback(int feedbackId):
a. Retrieve the existing feedback from the database with the specified **feedbackId**.
b. If no feedback with the specified feedbackId is found, return **false**.
c. If found, delete the feedback with the provided feedbackId.
d. Save changes asynchronously to the database.
e. 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):
a. Check if the email already exists in the database. If so return "**User already exists**".
b. Registers a new user with the provided details and assigns a role.
c. If any error occurs return "**User creation failed! Please check user details and try again**".
d. Return "**User created successfully!"** for the successful register.

2. public async Task<(int, string)> Login (LoginModel model):
a. Find user by email in the database.
b. Check if user exists, if not return "**Invalid email**".
c. If the user exists, check the password is correct, if not return "**Invalid password**".
d. Logs in a user with the provided credentials and generates a **JWT** token for authentication.

3. private string GenerateToken(IEnumerable<Claim> claims):
a. 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, string)> Login (LoginModel model);

Controllers:
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.

FeedController (Controllers / FeedController.cs):
This controller manages **feeds**, interacting with the **FeedService** to perform CRUD operations.

Functions:
1. public async Task<ActionResult<IEnumerable<Feed>>> GetAllFeeds()
a. Retrieves and returns all feeds from the database.
b. It calls the _**feedService.GetAllFeeds()** method to fetch all feeds from the service layer.
c. Returns a 200 OK response with the retrieved feed applications.
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<Feed>> GetFeedById(int feedId)
a. Retrieves a feed from the database with the specified feedId.
b. It calls the _**feedService.GetFeedById(feedId)** method to retrieve the feed from the service layer.
c. If the feed is not found, it returns a 404 Not Found response with a message "**Cannot find any feed**".
d. If the feed is found, it returns a 200 OK response with the feed data.
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<IEnumerable<Feed>>> GetFeedsByUserId(int userId)
a. Retrieves and returns all feeds from the database with the specified userId.
b. It calls the _**feedService.GetFeedsByUserId(userId)** method to fetch feeds associated with the specified userId.
c. If no feeds are found, it returns a 404 Not Found response with a message "Cannot find any feeds for this user".
d. If feeds are found, it returns a 200 OK response with the feed data.
e. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> AddFeed([FromBody] Feed feed)
a. Adds a new feed to the database.
b. It receives the feed data in the request body.
c. It tries to add the feed using the _**feedService.AddFeed(feed)** method.
d. If adding the feed is successful, it returns a 200 OK response with a success message "Feed added successfully".
e. If adding the feed fails, it returns a 500 Internal Server Error response with a failure message "Failed to add feed".
f. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> UpdateFeed(int feedId, [FromBody] Feed feed)

a. Updates an existing feed in the database.
b. It receives the feed ID and updated feed data in the request body.
c. It tries to update the feed using the **_feedService.UpdateFeed(feedId, feed)** method.
d. If the update is successful, it returns a 200 OK response with a success message “Feed updated successfully”.
e. If the feed is not found, it returns a 404 Not Found response with a message “Cannot find any feed”.
f. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

6. public async Task<ActionResult> DeleteFeed(int feedId)
a. Deletes a feed from the database.
b. It receives the feed ID to be deleted.
c. It tries to delete the feed using the **_feedService.DeleteFeed(feedId)** method.
d. If the deletion is successful, it returns a 200 OK response with a success message “Feed deleted successfully”.
e. If the feed is not found, it returns a 404 Not Found response with a message “**Cannot find any feed**”.
f. If an exception occurs during the process, it returns a 500 Internal Server Error response with the exception message.

LivestockController (Controllers / LivestockController .cs):
This controller manages **livestock**, interacting with the **LivestockService** to perform CRUD operations.

Functions:
1. public async Task<ActionResult<IEnumerable<Livestock>>> GetAllLivestocks()
a. Retrieves and returns all livestock from the database.
b. Calls the _livestockService.GetAllLivestocks() method to fetch all livestock from the service layer.
c. Returns a 200 OK response with the retrieved livestock list.

2. public async Task<ActionResult<Livestock>> GetLivestockById(int livestockId)
a. Retrieves a livestock from the database with the specified livestockId.
b. Calls the **_livestockService.GetLivestockById(livestockId)** method to retrieve the livestock from the service layer.
c. If the livestock is not found, returns a 404 Not Found response with a message "Cannot find any livestock".
d. If the livestock is found, returns a 200 OK response with the livestock data.
e. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

3. public async Task<ActionResult<IEnumerable<Livestock>>> GetLivestocksByUserId(int userId)
a. Retrieves and returns all livestock associated with the specified userId from the database.
b. Calls the **_livestockService.GetLivestocksByUserId(userId)** method to fetch livestock associated with the specified userId from the service layer.
c. If no livestock is found, returns a 404 Not Found response with a message "**No livestock found for this user**".
d. If livestock is found, returns a 200 OK response with the livestock data.
e. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> AddLivestock([FromBody] Livestock livestock)
a. Adds a new livestock to the database.
b. Receives the livestock data in the request body.
c. Tries to add the livestock using the **_livestockService.AddLivestock(livestock)** method.
d. If adding the livestock is successful, returns a 200 OK response with a success message "**Livestock added successfully**".
e. If adding the livestock fails, returns a 500 Internal Server Error response with a failure message "Failed to add livestock".

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

5. public async Task<ActionResult> UpdateLivestock(int livestockId, [FromBody] Livestock livestock)
a. Updates an existing livestock in the database.
b. Receives the livestock ID and updated livestock data in the request body.
c. Tries to update the livestock using the _livestockService.UpdateLivestock(livestockId, livestock) method.
d. If the update is successful, returns a 200 OK response with a success message "Livestock updated successfully".
e. If the livestock is not found, returns a 404 Not Found response with a message "Cannot find any livestock".
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

6. public async Task<ActionResult> DeleteLivestock(int livestockId)
a. Deletes a livestock from the database.
b. Receives the livestock ID to be deleted.
c. Tries to delete the livestock using the _livestockService.DeleteLivestock(livestockId) method.
d. If the deletion is successful, returns a 200 OK response with a success message "Livestock deleted successfully".
e. If the livestock is not found, returns a 404 Not Found response with a message "Cannot find any livestock".
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

RequestController (Controllers/RequestController.cs):
This controller manages requests, interacting with the **RequestService** to perform CRUD operations.

Functions:
1. public async Task<ActionResult<IEnumerable<Request>>> GetAllRequests()
a. Retrieves and returns all requests from the database.
b. Calls the _**requestService.GetAllRequests()** method to fetch all requests.
c. Returns a b. 200 OK response with the retrieved list of requests.
d. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

2. public async Task<ActionResult<Request>> GetRequestById(int requestId)
a. Retrieves a specific request by its ID from the database.
b. Calls the _**requestService.GetRequestById(requestId)** method to retrieve the request by the provided ID.
c. If the request is not found, returns a 404 Not Found response with a message indicating that no request was found.
d. If the request is found, returns a 200 OK response with the request data.
e. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

3. public async Task<ActionResult<IEnumerable<Request>>> GetRequestsByUserId(int userId)
a. Retrieves all requests associated with a specific user ID from the database.
b. Calls the _**requestService.GetRequestsByUserId(userId)** method to fetch requests for the specified user.
c. If no requests are found for the user, returns a 404 Not Found response with a message indicating that no requests were found.
d. If requests are found, returns a 200 OK response with the list of requests.
e. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult<IEnumerable<Request>>> GetRequestsByUserIdInMedicineOrFeed(int userId)
a. Retrieves and returns all requests associated with a specific user ID in medicine or feed category from the database.
b. Calls the _**requestService.GetRequestsByUserIdInMedicineOrFeed(userId)** method to fetch requests for the c. specified user in the medicine or feed category.
d. If no requests are found for the specified user, returns a 404 Not Found response with a message indicating that no requests were found.
e. If requests are found, returns a 200 OK response with the list of requests.

5. public async Task<ActionResult> AddRequest([FromBody] Request request)
a. Adds a new request to the database.
b. Receives the request data in the request body.
c. Attempts to add the request using the _**requestService.AddRequest(request)** method.
d. If adding the request is successful, returns a 200 OK response with a success message "**Request added successfully**".
e. If adding the request fails, returns a 500 Internal Server Error response with a failure message "Failed to add request".
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

6. public async Task<ActionResult> UpdateRequest(int requestId, [FromBody] Request request)
a. Updates an existing request in the database.
b. Receives the request ID and updated request data in the request body.
c. Tries to update the request using the _**requestService.UpdateRequest(requestId, request)** method.
d. If the update is successful, returns a 200 OK response with a success message "Request updated successfully".
e. If the request is not found, returns a 404 Not Found response with a message indicating that no request was found.
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

7. public async Task<ActionResult> DeleteRequest(int requestId)
a. Deletes a request from the database.
b. Receives the request ID to be deleted.
c. Tries to delete the request using the _**requestService.DeleteRequest(requestId)** method.
d. If the deletion is successful, returns a 200 OK response with a success message "Request deleted successfully".
e. If the request is not found, returns a 404 Not Found response with a message indicating that no request was found.
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

MedicineController (Controllers / MedicineController.cs):
This controller manages **medicine**, interacting with the **MedicineService** to perform CRUD operations.

Functions:
1. public async Task<ActionResult<IEnumerable<Medicine>>> GetAllMedicines()
a. Retrieves and returns all medicines from the database.
b. Calls the _**medicineService.GetAllMedicines()** method to fetch all medicines from the service layer.
c. Returns a 200 OK response with the retrieved medicines list.
d. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

2. public async Task<ActionResult<Medicine>> GetMedicineById(int medicineId)

a. Retrieves a medicine from the database with the specified medicineId.
b. Calls the **_medicineService.GetMedicineById(medicineId)** method to retrieve the medicine from the service layer.
c. If the medicine is not found, returns a 404 Not Found response with a message "Cannot find any medicine".
d. If the medicine is found, returns a 200 OK response with the medicine data.
e. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

3. public async Task<ActionResult> AddMedicine([FromBody] Medicine medicine)
a. Adds a new medicine to the database.
b. Receives the medicine data in the request body.
c. Tries to add the medicine using the **_medicineService.AddMedicine(medicine)** method.
d. If adding the medicine is successful, returns a 200 OK response with a success message "Medicine added successfully".
e. If adding the medicine fails, returns a 500 Internal Server Error response with a failure message "Failed to add medicine".
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdateMedicine(int medicineId, [FromBody] Medicine medicine)
a. Updates an existing medicine in the database.
b. Receives the medicine ID and updated medicine data in the request body.
c. Tries to update the medicine using the **_medicineService.UpdateMedicine(medicineId, medicine)** method.
d. If the update is successful, returns a 200 OK response with a success message "Medicine updated successfully".
e. If the medicine is not found, returns a 404 Not Found response with a message "Cannot find any medicine".
f. If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult<IEnumerable<Medicine>>> GetMedicinesByUserId(int userId)
a. Retrieves and returns all medicines associated with the specified userId from the database.
b. Calls the **_medicineService.GetMedicinesByUserId(userId)** method to fetch medicines associated with the specified userId from the service layer.
c. If no medicines are found, returns a 404 Not Found response with a message "Cannot find any medicines for this user".
d. If medicines are found, returns a 200 OK response with the medicines data.

6. public async Task<ActionResult> DeleteMedicine(int medicineId)
a. Deletes a medicine from the database.
b. Receives the medicine ID to be deleted.
c. Tries to delete the medicine using the **_medicineService.DeleteMedicine(medicineId)** method.
d. If the deletion is successful, returns a 200 OK response with a success message "Medicine deleted successfully".
e. If the medicine is not found, returns a 404 Not Found response with a message "Cannot find any medicine".
f. If an exception occurs during the process, 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.**

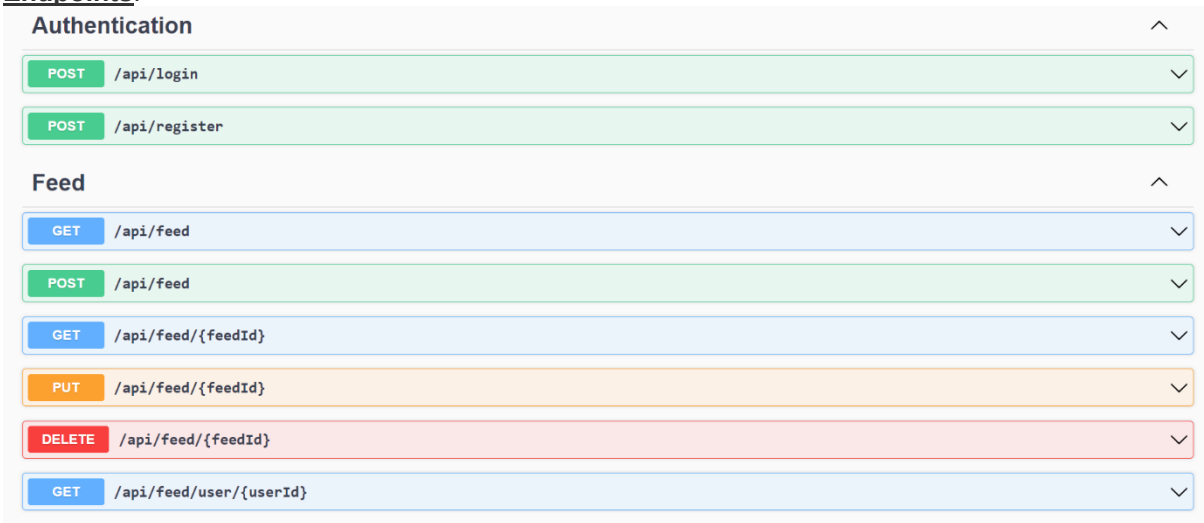
b. The **GetAllFeedbacks** method is a controller action responsible for retrieving all feedbacks.
c. It tries to get all feedbacks using the **_feedbackService.GetAllFeedbacks()** method.
d. If the operation is successful, it returns a **200 OK response** with the retrieved feedbacks.
e. If an exception occurs during the process, it returns a **500 Internal Server Error response** with the exception message.

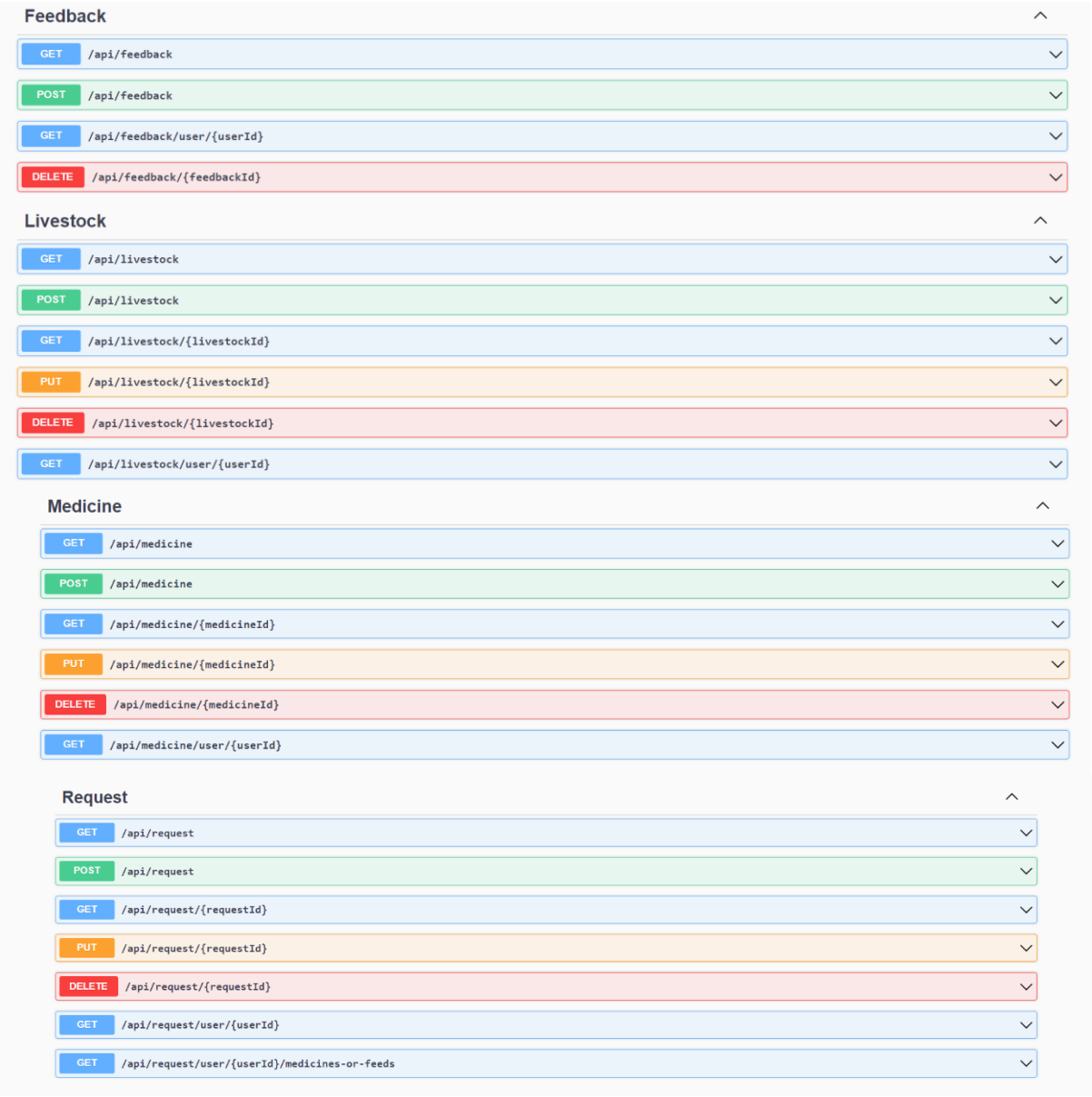
2. public async Task<ActionResult<IEnumerable<Feedback>>> GetFeedbacksByUserId(int userId):
a. Implement the logic inside **try-catch block**.
b. The **GetFeedbacksByUserId** method is a controller action responsible for retrieving feedbacks by **userId**.
c. It tries to get feedbacks by userId using 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:





1. Login: [Access for both Owner and Supplier]

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 Owner and Supplier]

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 feeds: [Access for Owner]

Endpoint name: "/api/feed"
Method: GET
Response:

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

4. Post feed: [Access for Supplier]

Endpoint name: "/api/feed"
Method: POST
Request body:
{
"FeedName": "string",
"Type": "string",
"Description": "string",
"Quantity": 0,
"Unit": "string",

"PricePerUnit": 0,
"Image": "string",
"UserId": 0
}
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.

5. Get specific feed: [Access for only Supplier]

Endpoint name: "/api/feed/{feedId}"
Method: GET
Parameter: feedId
Response:

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

6. Update feed: [Access for only Supplier]

Endpoint name: "/api/feed/{feedId}"
Method: PUT
Parameter: feedId
Request body:
{
"FeedName": "string",
"Type": "string",
"Description": "string",
"Quantity": 0,
"Unit": "string",
"PricePerUnit": 0,
"Image": "string",
"UserId": 0
}
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.

7. Delete feed: [Access for only Supplier]

Endpoint name: "/api/feed/{feedId}"
Method: DELETE
Parameter: feedId

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.

8. Get feeds specific to user: [Access for only Supplier]

Endpoint name: "/api/feed/user/{userId }"
Method: GET
Parameter: userId
Response:

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

9. Get all medicines: [Access for Owner]

Endpoint name: "/api/medicine"
Method: GET
Response:

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

10. Post medicine: [Access for Supplier]

Endpoint name: "/api/medicine"
Method: POST
Request body:

```
{
"MedicineName": "string",
"Brand": "string",
"Category": "string",
"Description": "string",
"Quantity": 0,
"Unit": "string",
"PricePerUnit": 0,
"Image": "string",
"UserId": 0,
}
```

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.

11. Get specific medicine: [Access for only Supplier]

Endpoint name: “/api/medicine/{medicineId}”
Method: GET
Parameter: medicineId
Response:

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

12. Update medicine: [Access for only Supplier]

Endpoint name: “/api/medicine/{medicineId}”
Method: PUT
Parameter: medicineId
Request body:

```
{
"MedicineName": "string",
"Brand": "string",
"Category": "string",
"Description": "string",
"Quantity": 0,
"Unit": "string",
"PricePerUnit": 0,
"Image": "string",
"UserId": 0,
}
```

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.

13. Delete medicine: [Access for only Supplier]

Endpoint name: “/api/medicine/{medicineId}”
Method: DELETE
Parameter: medicineId
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.

14. Get medicines specific to user: [Access for only Supplier]

Endpoint name: “/api/medicine/user/{userId }”
Method: GET

Parameter: userId
Response:

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

15. Post livestock: [Access for Owner]

Endpoint name: “/api/livestock”
Method: POST
Request body:
{
"Name": "string",
"Species": "string",
"Age": 0,
"Breed": "string",
"HealthCondition": "string",
"Location": "string",
"VaccinationStatus": "string",
"UserId": 0
}
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.

16. Get specific livestock: [Access for only Owner]

Endpoint name: “/api/livestock/{livestockId}”
Method: GET
Parameter: livestockId
Response:

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

17. Update livestock: [Access for only Owner]

Endpoint name: “/api/livestock/{livestockId}”
Method: PUT
Parameter: livestockId
Request body:
{
"Name": "string",
"Species": "string",
"Age": 0,
"Breed": "string",
"HealthCondition": "string",

"Location": "string",
"VaccinationStatus": "string",
"UserId": 0
}
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. Delete livestock: [Access for only Owner]

Endpoint name: “/api/livestock/{livestockId}”
Method: DELETE
Parameter: livestockId
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.

19. Get livestocks specific to user: [Access for only Owner]

Endpoint name: “/api/livestock/user/{userId}”
Method: GET
Parameter: userId
Response:

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

20. Post request: [Access for Owner]

Endpoint name: “/api/request”
Method: POST
Request body:
{
"RequestType": "string",
"FeedId": 0,
"MedicineId": 0,
"UserId": 0,
"LivestockId": 0,
"Quantity": 0,
"Status": "string",
"RequestDate": "2024-05-20T11:46:25.512Z"
}

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.

21. Update request: [Access for only Supplier]

Endpoint name: “/api/request/{requestId}”
Method: PUT
Parameter: requestId
Request body:
{
"RequestType": "string",
"FeedId": 0,
"MedicineId": 0,
"UserId": 0,
"LivestockId": 0,
"Quantity": 0,
"Status": "string",
"RequestDate": "2024-05-20T11:46:25.512Z"
}
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.

22. Delete request: [Access for only Owner]

Endpoint name: “/api/request/{requestId}”
Method: DELETE
Parameter: requestId
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.

23. Get requests specific to user: [Access for only Owner]

Endpoint name: “/api/request/{requestId}”
Method: GET
Parameter: requestId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing requests details specific to user.
500	JSON object containing Error message.

24. Get all feedbacks: [Access for only Supplier]

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.

25. Add feedback: [Access for only Owner]

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.

26. Get feedback specific to user: [Access for only Owner]

Endpoint name: “/api/feedback/{userId }”
Method: GET
Parameter: userId
Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing feedback details specific to user.
500	JSON object containing Error message.

27. Delete feedback: [Access for only Owner]

Endpoint name: “/api/feedback /{feedbackId}”
Method: DELETE
Parameter: feedbackId

Response:

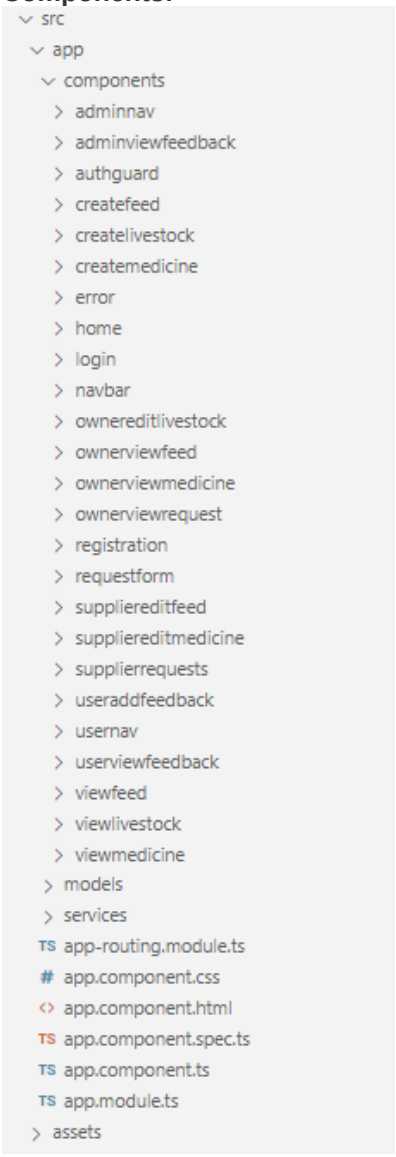
Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

Frontend Requirements:

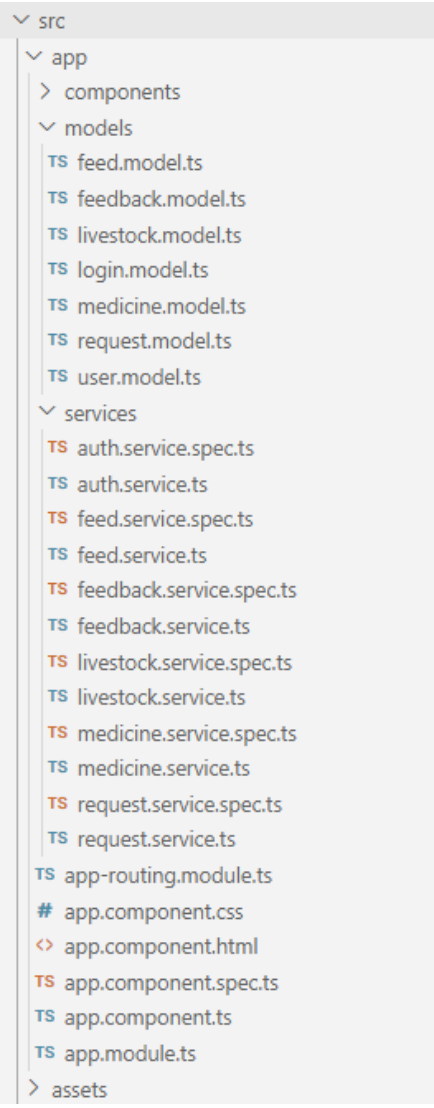
- Create a folder named **components** inside the app to store all the components. (Refer to project structure screenshots).
- Create a folder named **models** inside the app to store all the model interfaces.
- Create a folder named **services** inside the app to implement all the services.
- Create a model interface referring to the backend entities (User, Medicine, Feed, Livestock, Request, Feedback) 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:

```
export class User {
  UserId?: number;
  Email: string;
  Password: string;
  Username: string;
  MobileNumber: string;
  UserRole: string;
}
```

Login Model:

```
export class Login {
  Email: string;
  Password: string;
}
```

Medicine Model:

```
export interface Medicine {
  MedicineId?: number;
  MedicineName: string;
```

```
  Brand: string;
  Category: string;
  Description: string;
  Quantity: number;
  Unit: string;
  PricePerUnit: number;
  Image: string;
  UserId: number;
}
```

Feed Model:

```
export interface Feed {
  FeedId?: number;
  FeedName: string;
  Type: string;
  Description: string;
  Quantity: number;
  Unit: string;
  PricePerUnit: number;
  Image: string;
  UserId: number;
}
```

Livestock Model:

```
export interface Livestock {
  LivestockId: number;
  Name: string;
  Species: string;
  Age: number;
  Breed: string;
  HealthCondition?: string;
  Location: string;
  VaccinationStatus?: string;
  UserId: number;
}
```

Request Model:

```
export interface Request {
  RequestType: string;
  MedicineId: number;
  FeedId: number;
  UserId: number;
  Quantity: number;
  Status: string;
  LivestockId: number;
  RequestDate: string; // ISO 8601 formatted date string
}
```

Feedback Model:

```
  export 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.

1. AuthService(auth.service.ts):

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

Methods Overview:

- register(user: User): Observable<any>:
  - 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.

- login(login : Login): Observable<any>:
  - 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.

2. FeedService (feed.service.ts):

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

Methods Overview:

- addFeed(requestObject: Feed): Observable<Feed>:
  - Use this method to add a new feed. It sends a POST request to the '/api/feed' endpoint with the feed data provided as the body and the authorization token prefixed with 'Bearer' stored in localStorage.\

- getFeedByUserID(id: number): Observable<Feed>:
  - This method is used to get a feed by the user ID. It sends a GET request to the '/api/feed/user/{id}' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getFeedById(id: string): Observable<Feed>:
  - Use this method to get a feed by ID. It sends a GET request to the '/api/feed/{id}' endpoint with the feed ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getAllFeed(): Observable<Feed[]>:
  - This method is used to get all feeds. It sends a GET request to the '/api/feed' endpoint with the authorization token prefixed with 'Bearer' stored in localStorage.

- deleteFeed(feedId: string): Observable<void>:
  - Use this method to delete a feed. It sends a DELETE request to the '/api/feed/{feedId}' endpoint with the feed ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- updateFeed(id: string, requestObject: Feed): Observable<Feed>:
  - This method is used to update a feed. It sends a PUT request to the '/api/feed/{id}' endpoint with the feed ID provided as a parameter, the feed data provided as the body, and the authorization token prefixed with 'Bearer' stored in localStorage.

3. MedicineService (medicine.service.ts):

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

Methods Overview:

- addMedicine(requestObject: Medicine): Observable<Medicine>:
  - Use this method to add a new medicine. It sends a POST request to the '/api/medicine' endpoint with the medicine data provided as the body and the authorization token prefixed with 'Bearer' stored in localStorage.

- getMedicineByUserID(id: number): Observable<Medicine>:
    - This method is used to get a medicine by the user ID. It sends a GET request to the '/api/medicine/user/{id}' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getMedicineById(id: string): Observable<Medicine>:
    - Use this method to get a medicine by ID. It sends a GET request to the '/api/medicine/{id}' endpoint with the medicine ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getAllMedicine(): Observable<Medicine[]>:
    - This method is used to get all medicines. It sends a GET request to the '/api/medicine' endpoint with the authorization token prefixed with 'Bearer' stored in localStorage.

- deleteMedicine(medicineId: string): Observable<void>:
    - Use this method to delete a medicine. It sends a DELETE request to the '/api/medicine/{medicineId}' endpoint with the medicine ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- updateMedicine(id: string, requestObject: Medicine): Observable<Medicine>:
    - This method is used to update a medicine. It sends a PUT request to the '/api/medicine/{id}' endpoint with the medicine ID provided as a parameter, the medicine data provided as the body, and the authorization token prefixed with 'Bearer' stored in localStorage.

4. LivestockService (livestock.service.ts)

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

Methods Overview:

- getLivestockByUserID(id: number): Observable<Livestock>:
    - This method is used to get livestock by the user ID. It sends a GET request to the '/api/livestock/user/{id}' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getLivestockByID(id: number): Observable<Livestock>:
    - Use this method to get livestock by ID. It sends a GET request to the '/api/livestock/{id}' endpoint with the livestock ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- addLivestock(livestock: Livestock): Observable<Livestock>:
    - This method is used to add a new livestock. It sends a POST request to the '/api/livestock' endpoint with the livestock data provided as the body and the authorization token prefixed with 'Bearer' stored in localStorage.

- updateLivestock(id: number, livestock: Livestock): Observable<Livestock>:
    - Use this method to update a livestock. It sends a PUT request to the '/api/livestock/{id}' endpoint with the livestock ID provided as a parameter, the livestock data provided as the body, and the authorization token prefixed with 'Bearer' stored in localStorage.

- deleteLivestock(id: number): Observable<void>:
    - This method is used to delete a livestock. It sends a DELETE request to the '/api/livestock/{id}' endpoint with the livestock ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

5. RequestService (request.service.ts):

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

Methods Overview:

- addRequest(request: Request): Observable<Request>:
  - Use this method to add a new request. It sends a POST request to the '/api/request' endpoint with the request data provided as the body and the authorization token prefixed with 'Bearer' stored in localStorage.

- getRequestsByUserId(userId: string): Observable<Request[]>:
  - This method is used to get requests by user ID. It sends a GET request to the '/api/request/user/{userId}' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- deleteRequest(requestId: string): Observable<any>:
  - Use this method to delete a request. It sends a DELETE request to the '/api/request/{requestId}' endpoint with the request ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getRequestByMedicineOrFeedUserId(userId: string): Observable<Request[]>:
  - This method is used to get requests by medicines or feeds user ID (medicines or feeds). It sends a GET request to the '/api/request/user/{userId}/medicines-or-feeds' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- updateRequestStatus(requestId: number, status: Request): Observable<any>:
  - Use this method to update a request status. It sends a PUT request to the '/api/request/{requestId}' endpoint with the request ID provided as a parameter, the request status provided as the body, and the authorization token prefixed with 'Bearer' stored in localStorage.

6. FeedbackServic(feedback.service.ts):

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

Methods Overview:

- addFeedback(feedback: Feedback): Observable<Feedback>:
  - Use this method to add a new feedback. It sends a POST request to the '/api/feedback' endpoint with the feedback data provided as the body and the authorization token prefixed with 'Bearer' stored in localStorage.

- getAllfeedbacksByUserId(userId: string): Observable<Feedback[]>:
  - This method is used to get all feedbacks by user ID. It sends a GET request to the '/api/feedback/user/{userId}' endpoint with the user ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- deleteFeedback(feedbackId: string): Observable<void>:
  - Use this method to delete a feedback. It sends a DELETE request to the '/api/feedback/{feedbackId}' endpoint with the feedback ID provided as a parameter and the authorization token prefixed with 'Bearer' stored in localStorage.

- getFeedbacks(): Observable<Feedback[]>:
  - This method is used to get all feedbacks. 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.

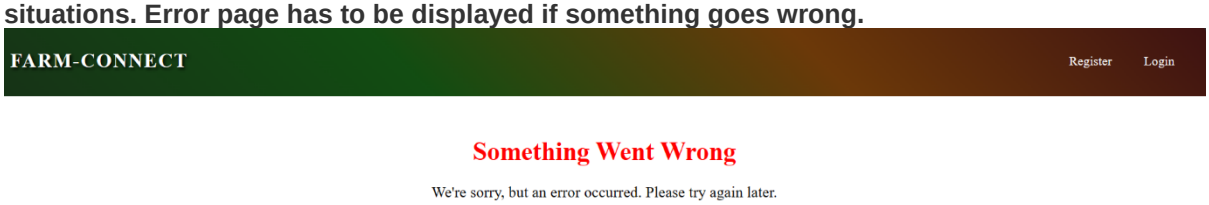
Server-Side Validation:

- 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.

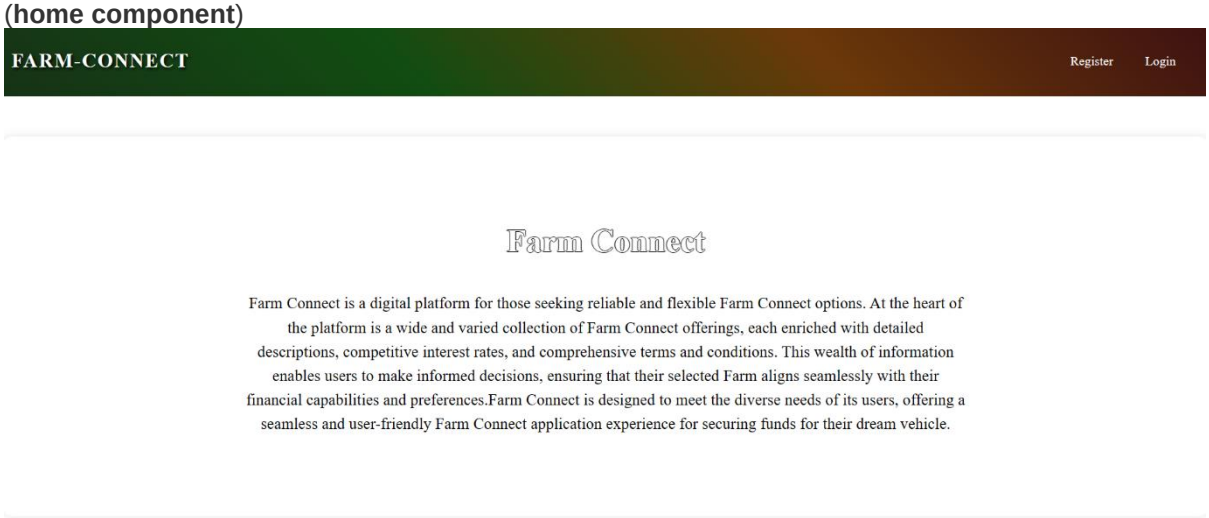


Frontend Screenshots:
• All the asterisk (*) marked fields are mandatory in the form. Make sure to mark all the field names with * symbol followed by the validations.

Navigation Bar:
(**navbar component**)

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

Landing Page:
(**home component**)

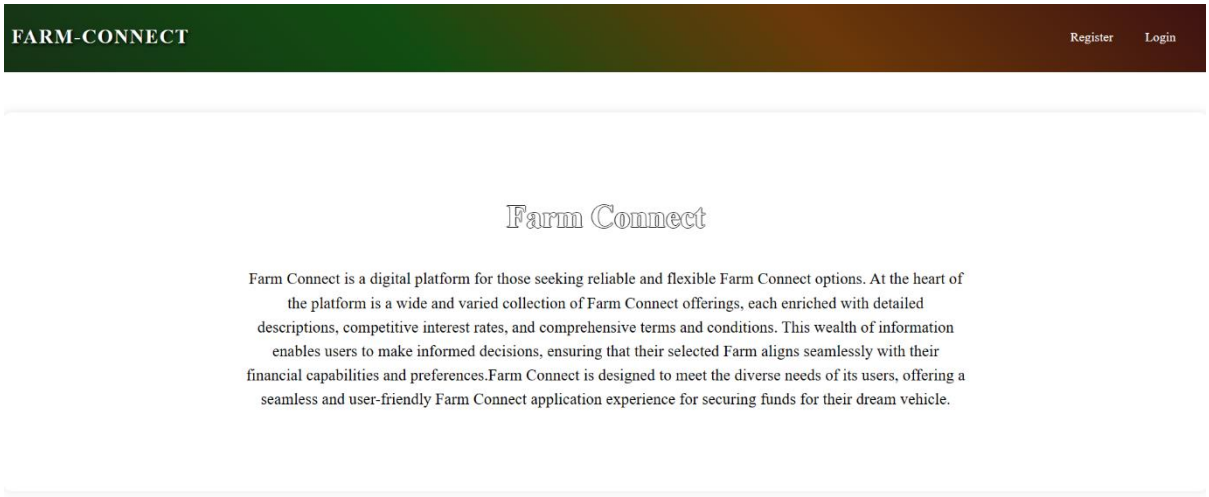


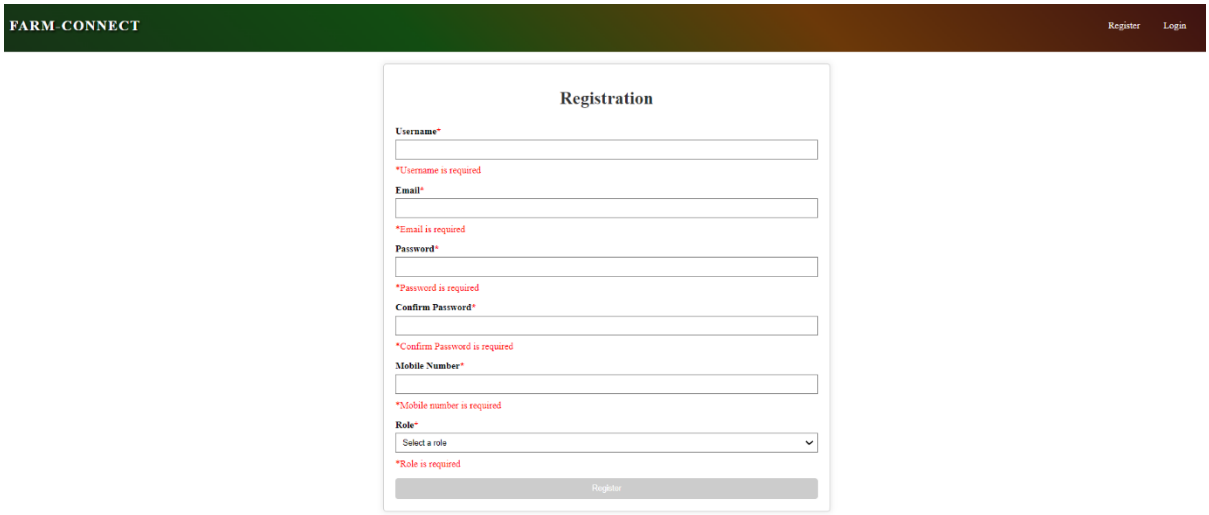
The component features a heading "**Farm Connect**" accompanied by an introductory message that provides an overview of the application.

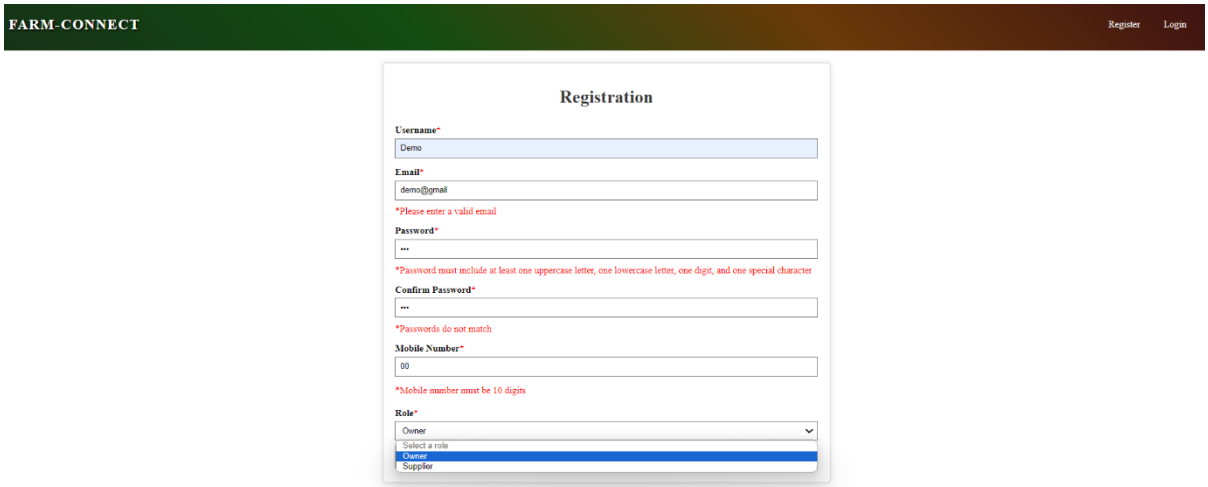
User side (Admin and User):

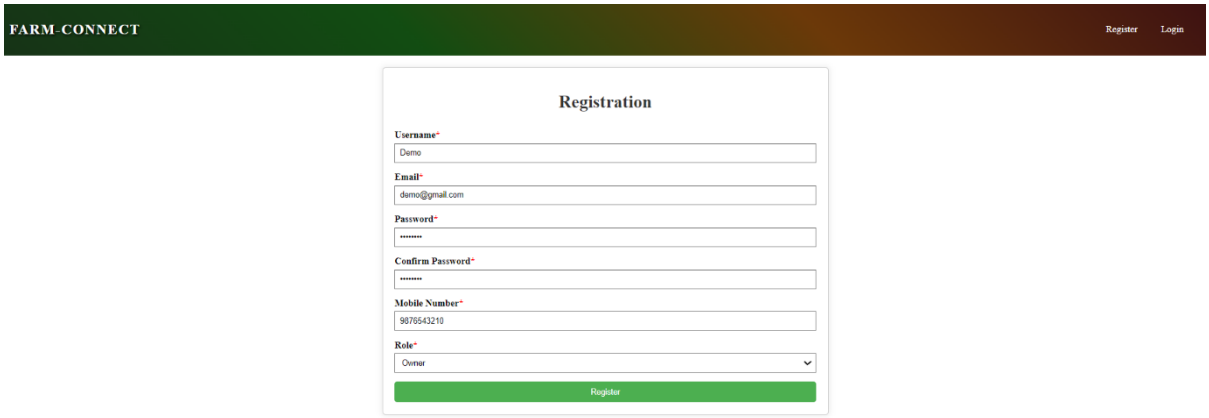
Registration Page:
(**registration component**)

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

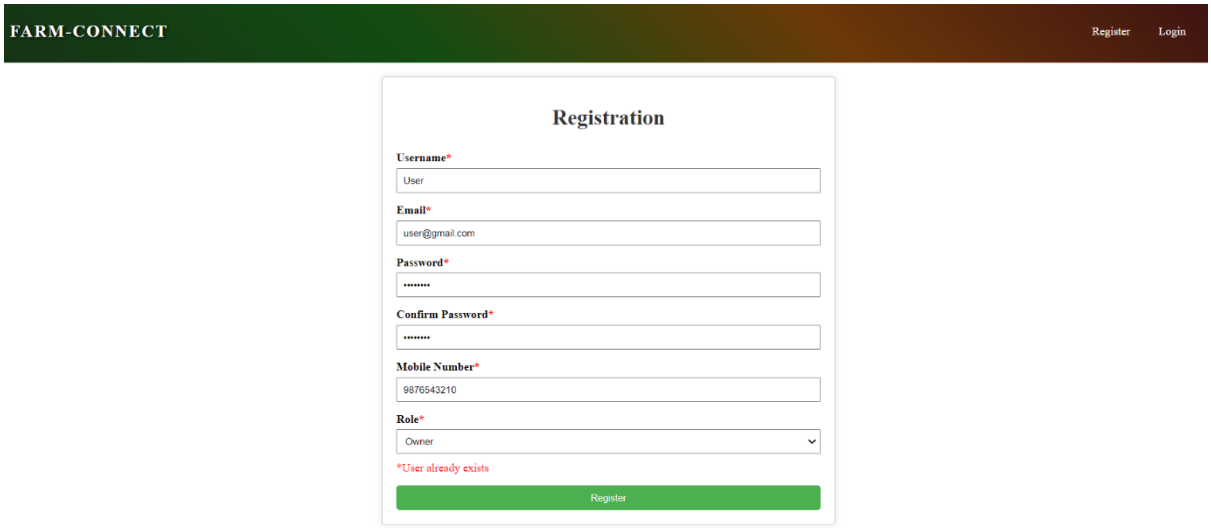








When the "**Register**" button is clicked, upon successful submission, the user must be navigated to the **login** page.

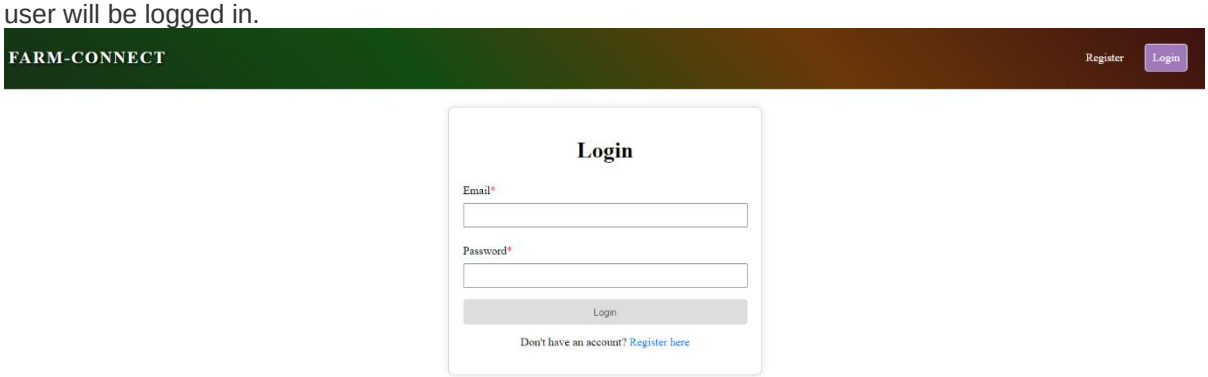


If a user attempts to register with an existing email, a message stating "User already exists" will be displayed.

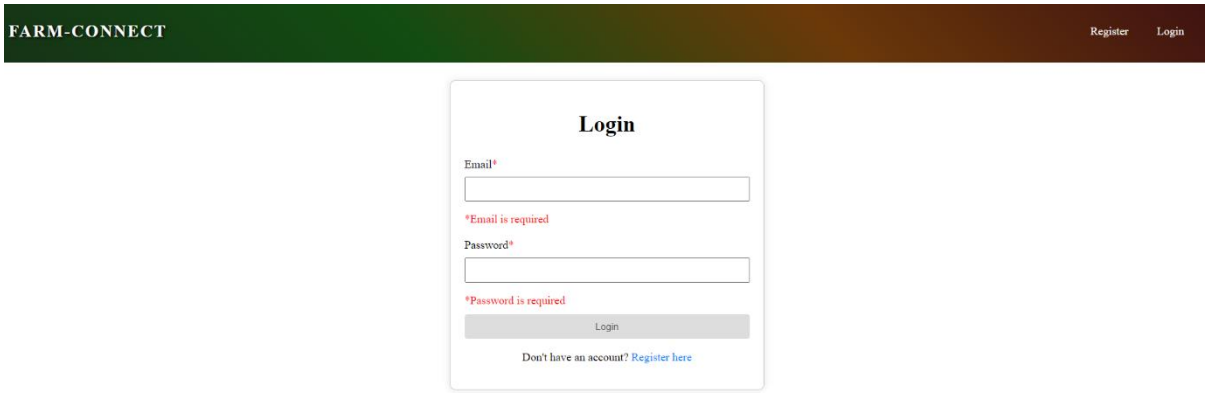
Login Page

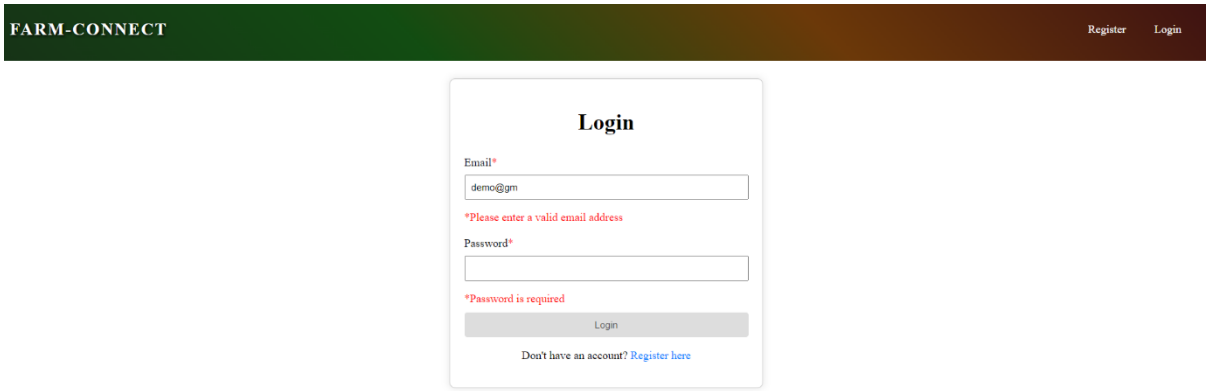
(**login component**)

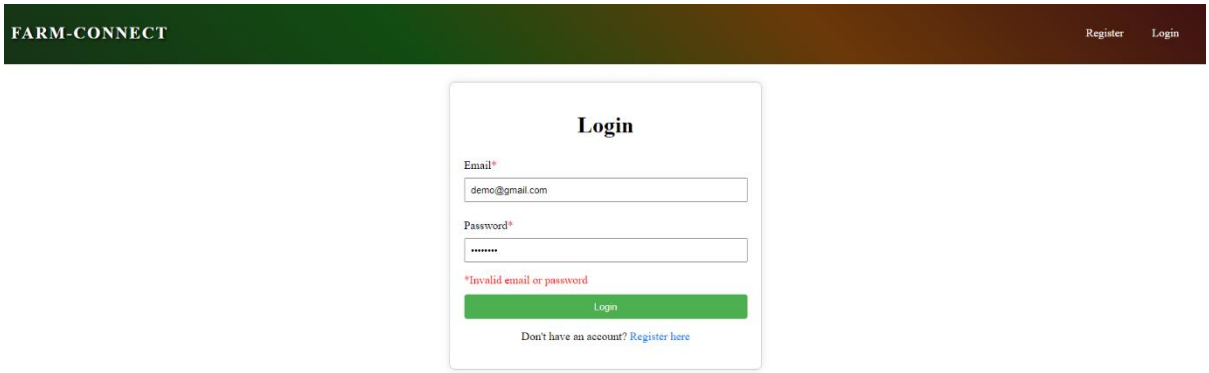
This page is used for logging in to the application. On providing the valid email and password, the user will be logged in.

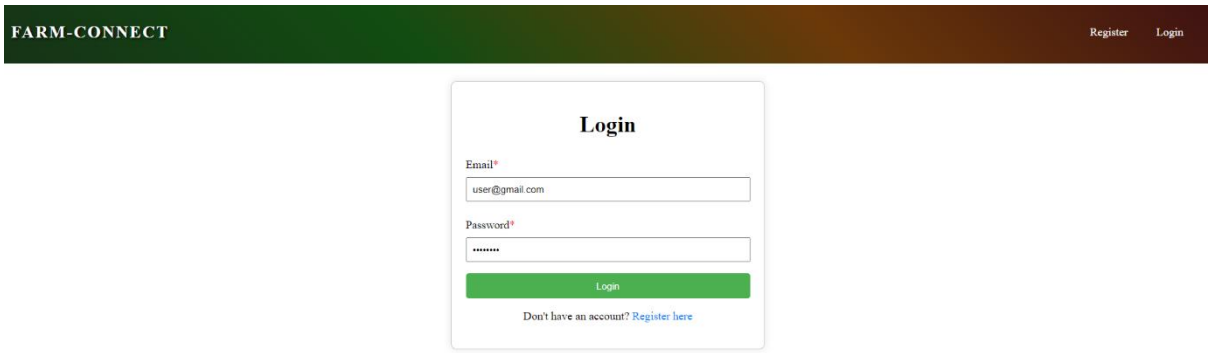


Perform validations for email and password fields.





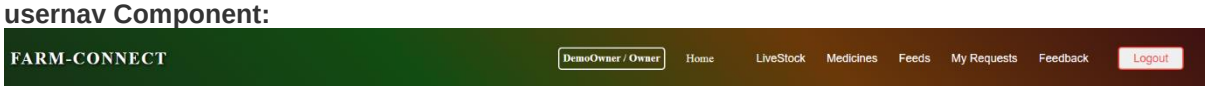




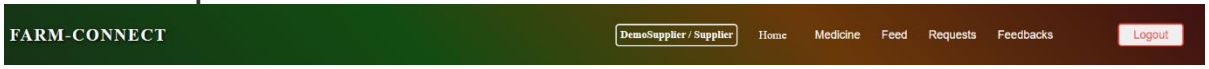
On Clicking the 'Login' button, user will be navigated to the respective pages with respective navbars (adminnav or usernav) based on their roles.

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

usernav Component:

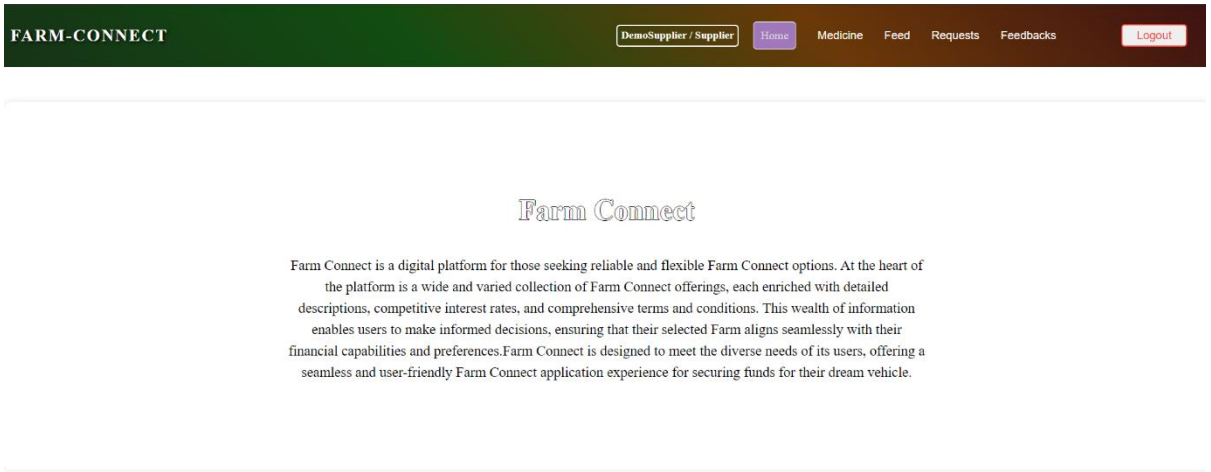


adminnav Component:



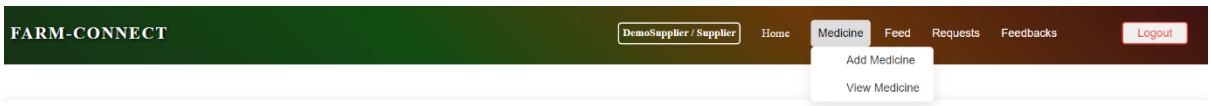
Suppliers Side (Admin Side):

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

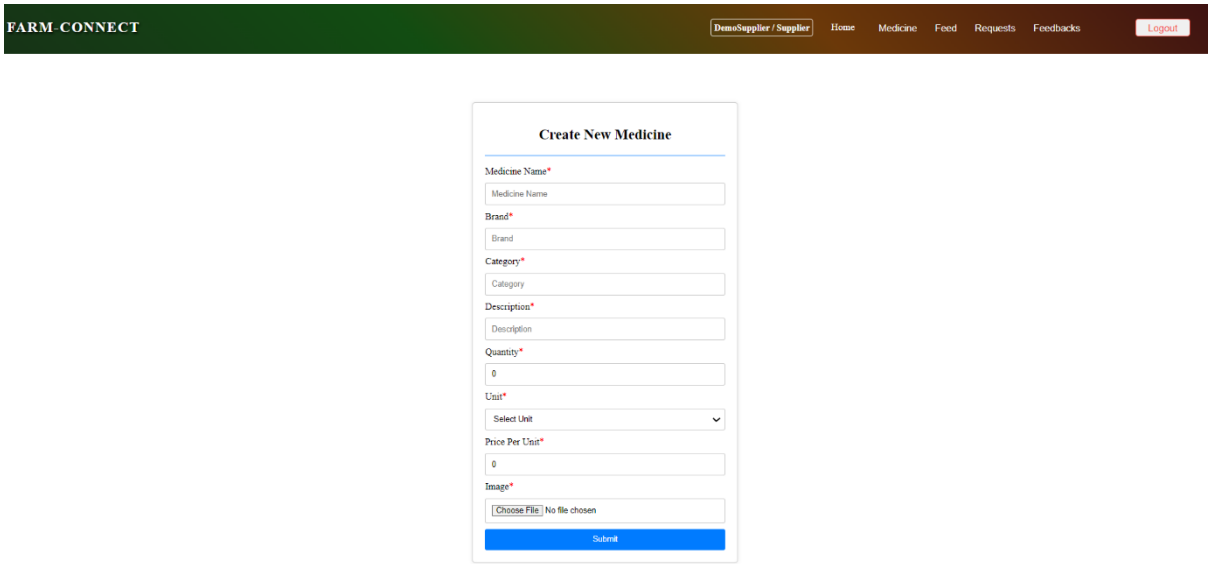


Suppliers (Admin) can navigate to other pages by clicking on the menu available in the navigation bar (**adminnav**).

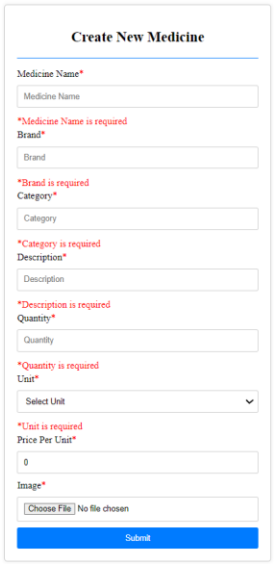
On hovering over the "**Medicine**" item in the navbar, a submenu should appear with options to "**Add Medicine**" and "**View Medicine**".



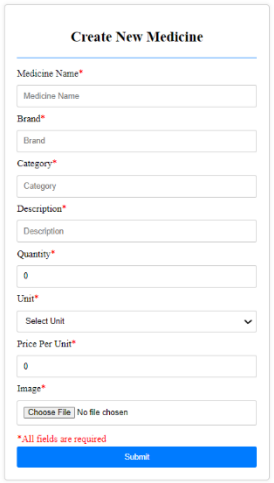
Clicking on "**Add Medicine**" will navigate to the **createmedicine component**, which displays a form with the heading "**Create New Medicine**". The Supplier (admin) should be able to add the Medicine details on this page, which is used to add a new Medicine to the system.

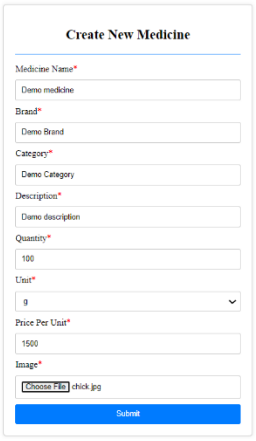


Perform validations for all the form fields.

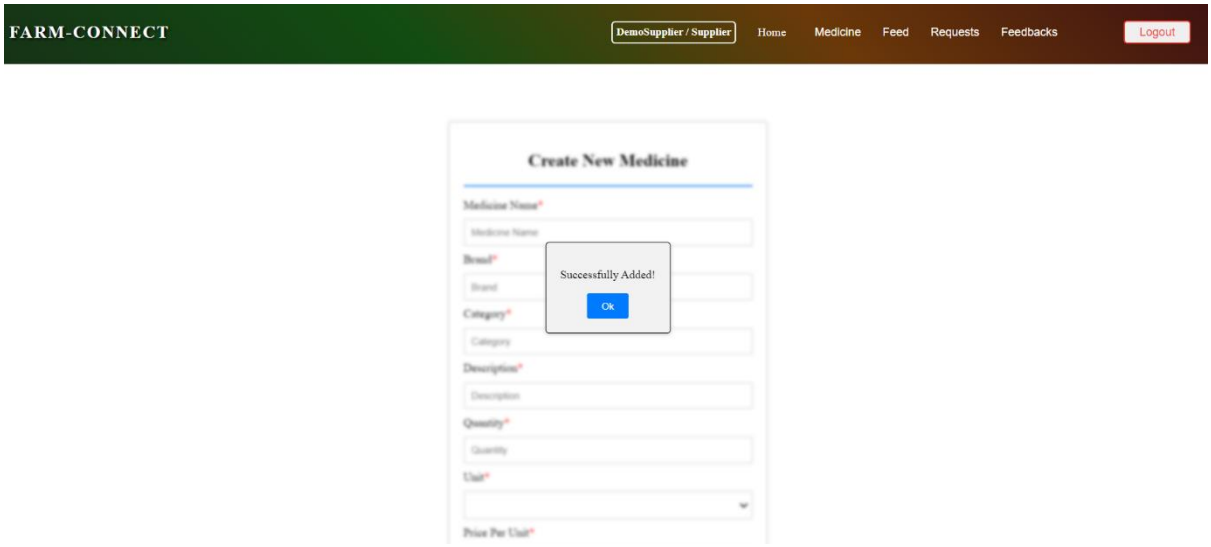


Clicking the "Submit" button with empty fields will display a validation message stating "All fields are required".

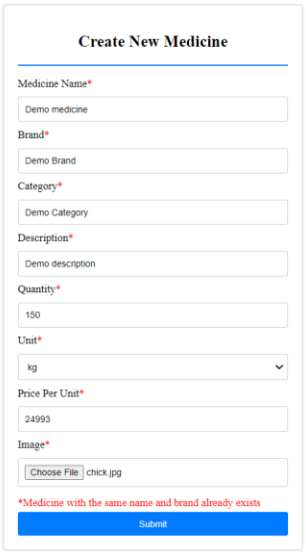




Upon clicking the "**Submit**" button, if the operation is successful, a popup message saying "**Successfully Added!**" should be displayed.



If the Supplier (admin) adds a new Medicine with an existing Medicine Name and Brand, an error message stating "**Medicine with the same name and brand already exists**" should be displayed.

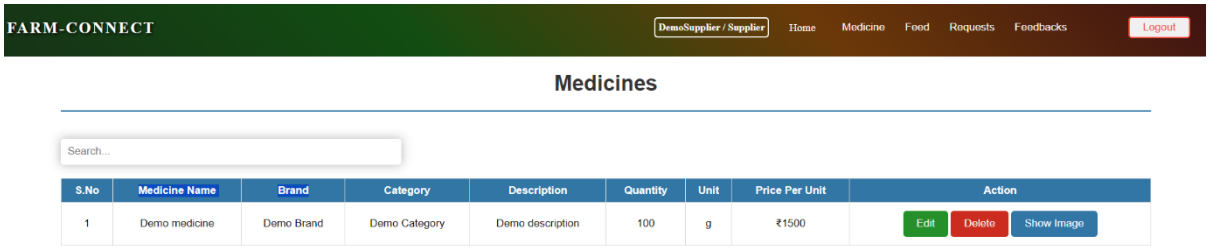


On Clicking the 'Submit' button a medicine is added to the system and supplier can add a new medicine again. To move to other pages supplier can click any of the menus available in the navbar.

Supplier View Medicine

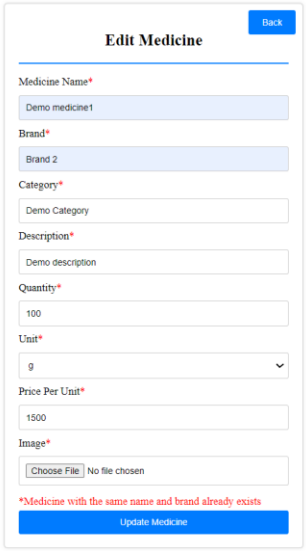
On hovering over the "**Medicine**" item in the navbar, a submenu should appear with options to "**Add Medicine**" and "**View Medicine**".

Clicking on "**View Medicine**" will navigate to the **viewmedicine component**, which displays all medicine details in a table format with the heading "**Medicines**". Additionally, a search feature is provided to search based on Medicine Name and Brand.

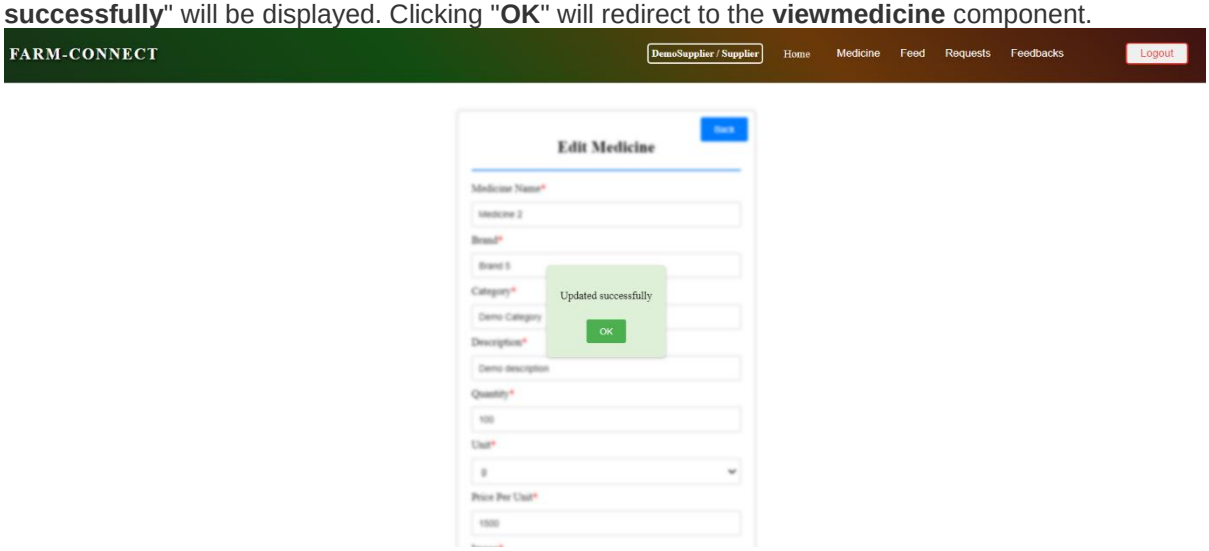


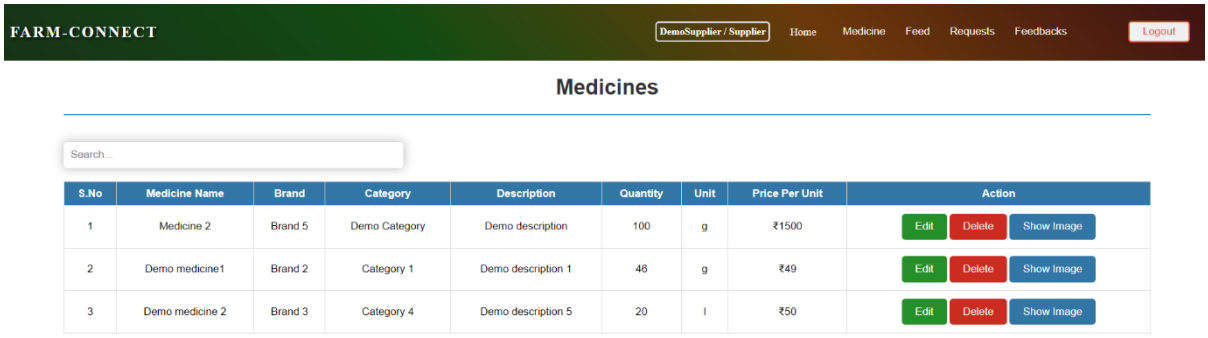
Clicking the edit button will navigate to the **supplierеditmedicine component**, which displays an editing form with pre-populated medicine data of the selected medicine. This form also contains a "Back" button that will navigate to the **viewmedicine component**. Validations are performed for all the form fields.

If the Supplier updates the medicine Name & brand to match an existing one, an error indicating "**Medicine with the same name and brand already exists**" will be displayed.

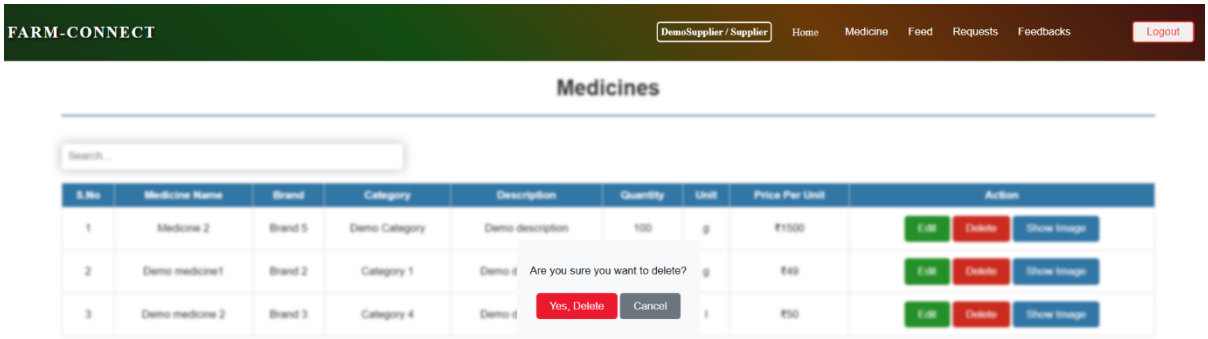


Upon clicking the "**Update Medicine**" button, if the operation is successful, a popup saying "**Updated successfully**" will be displayed. Clicking "**OK**" will redirect to the **viewmedicine** component.

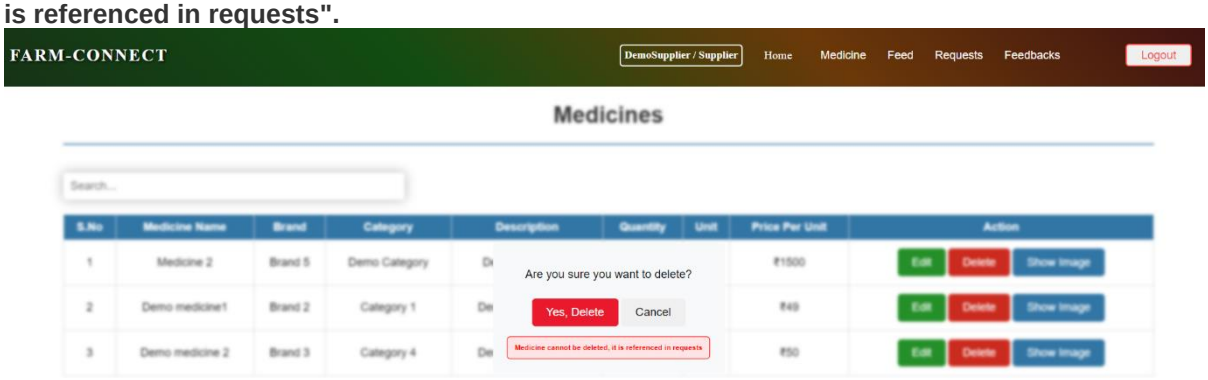




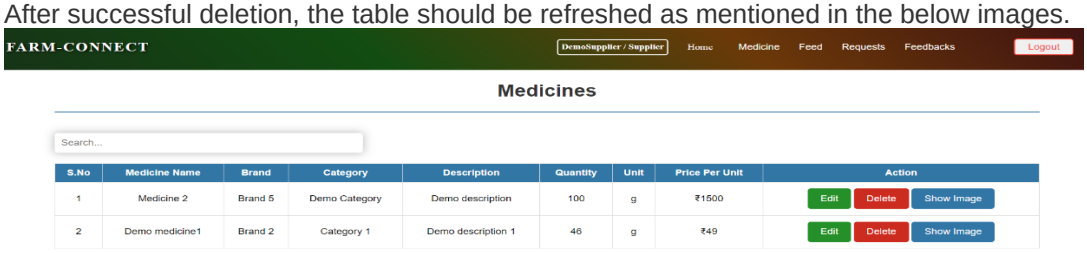
On clicking the "**Delete**" button, a pop-up should be displayed with a confirmatory message to delete the data.



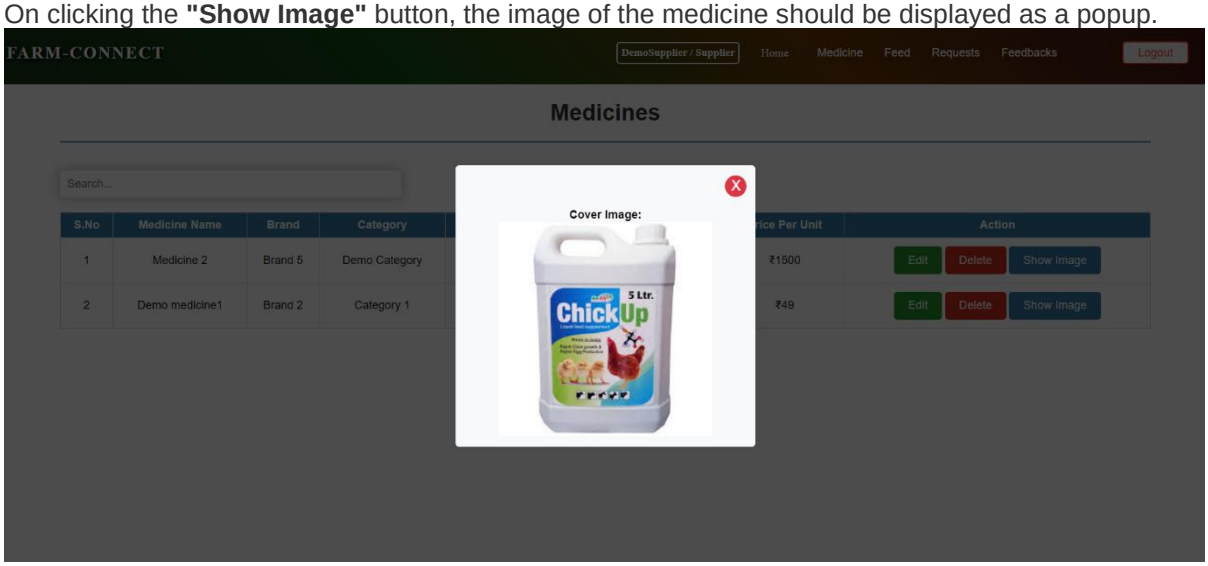
Once an Owner has requested for the medicine, the supplier should not be able to delete the medicine. Attempting to do so will display an error message stating **"Medicine cannot be deleted, it is referenced in requests".**



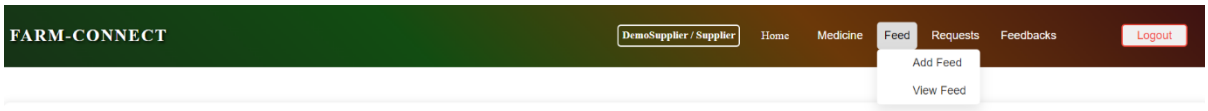
After successful deletion, the table should be refreshed as mentioned in the below images.



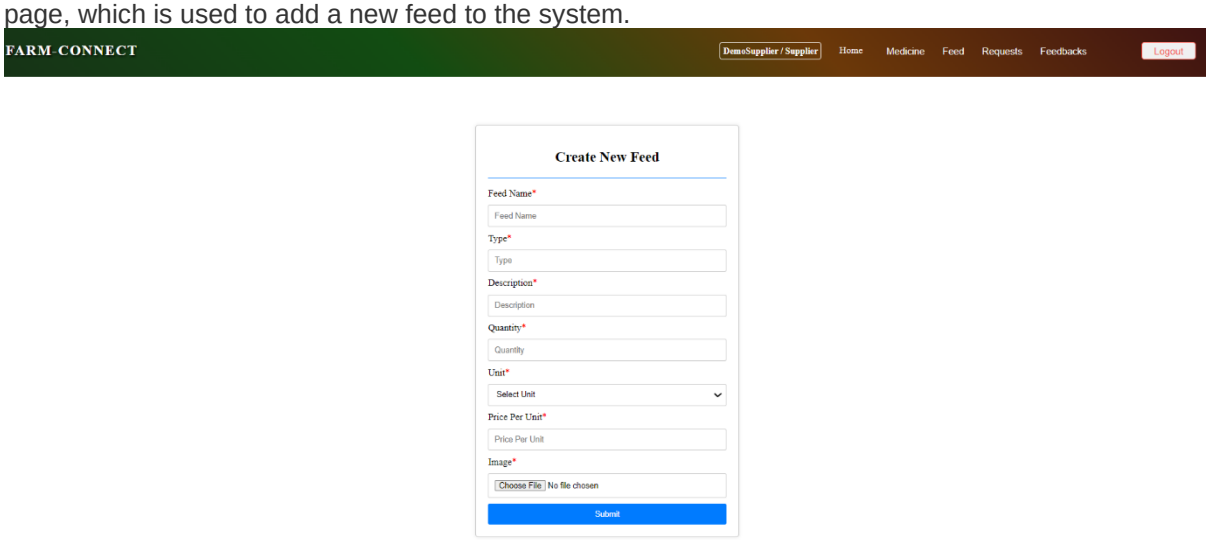
On clicking the **"Show Image"** button, the image of the medicine should be displayed as a popup.



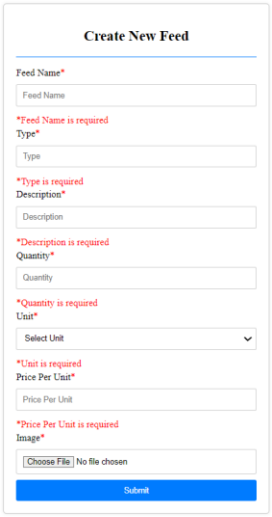
Supplier Feeds:
On hovering over the "**Feed**" item in the navbar, a submenu should appear with options to "**Add Feed**" and "**View Feed**".



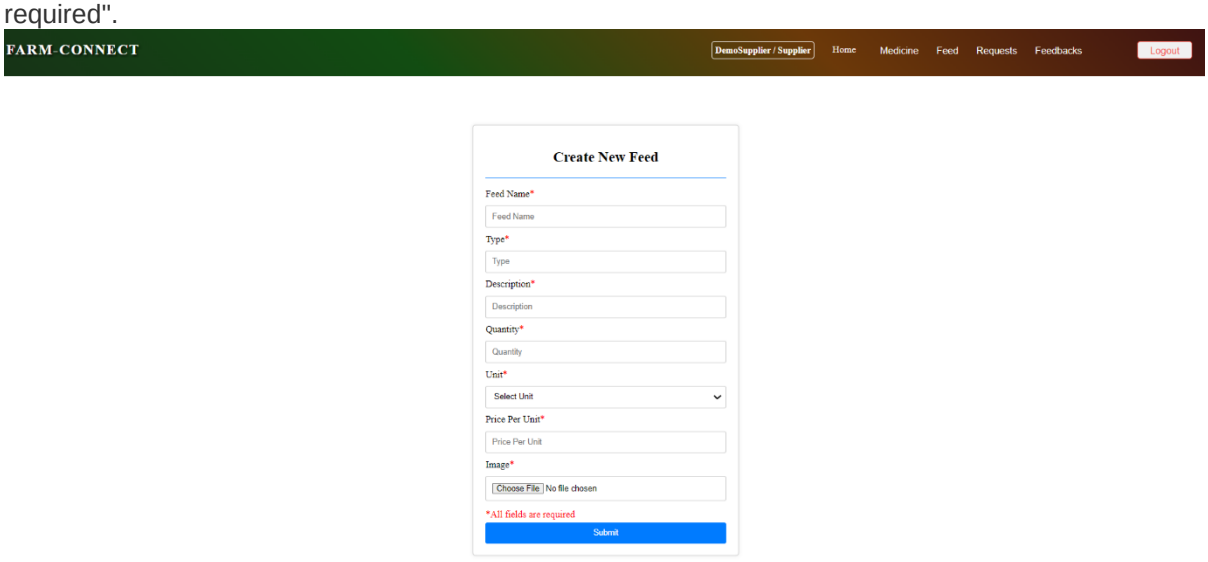
Clicking on "**Add Feed**" will navigate to the **createfeed component**, which displays a form with the heading "**Create New Feed**". The Supplier (admin) should be able to add the Feed details on this page, which is used to add a new feed to the system.

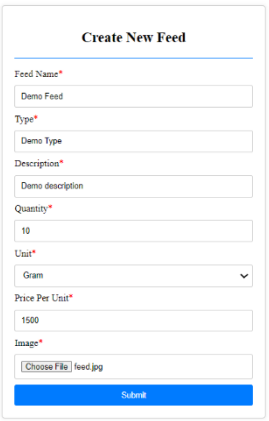


Perform validations for all the form fields.

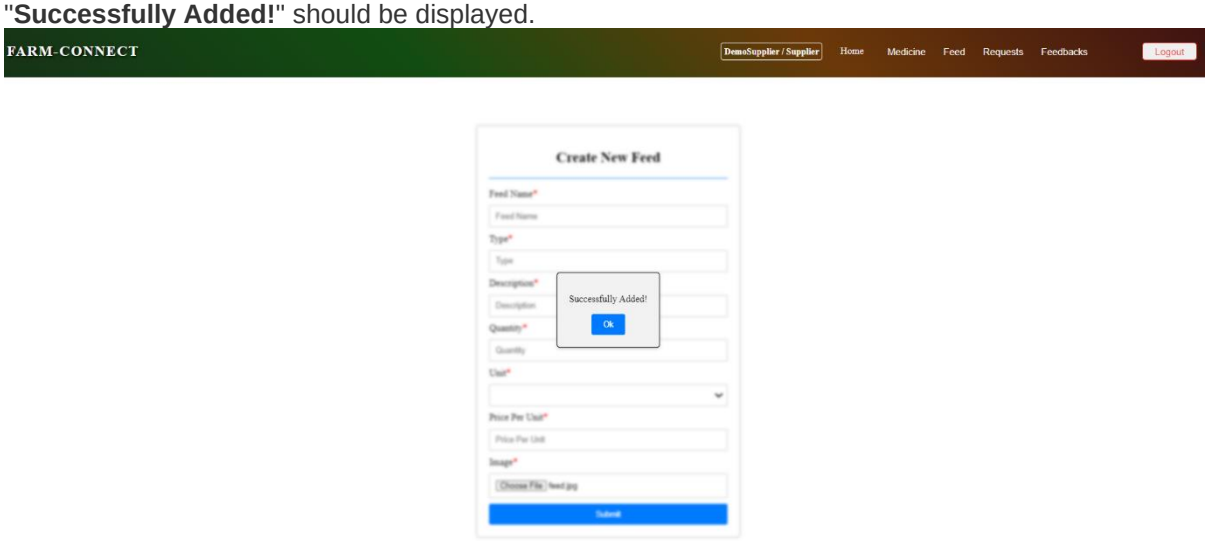


Clicking the "Submit" button with empty fields will display a validation message stating "All fields are required".

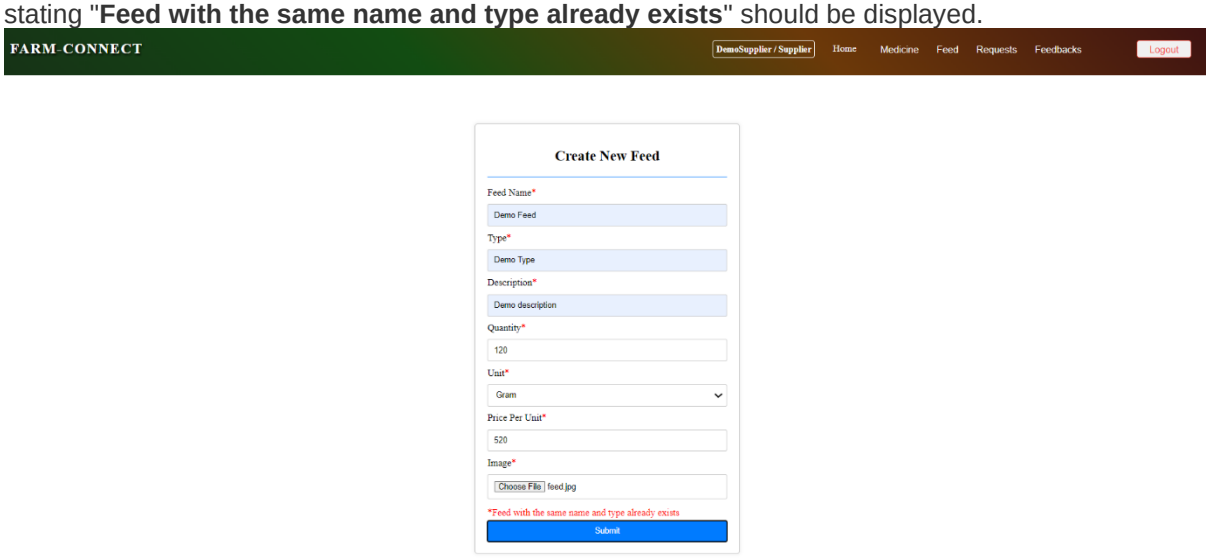




Upon clicking the "**Submit**" button, if the operation is successful, a popup message saying "**Successfully Added!**" should be displayed.



If the Supplier (admin) adds a new Feed with an existing Feed Name and Type, an error message stating "**Feed with the same name and type already exists**" should be displayed.

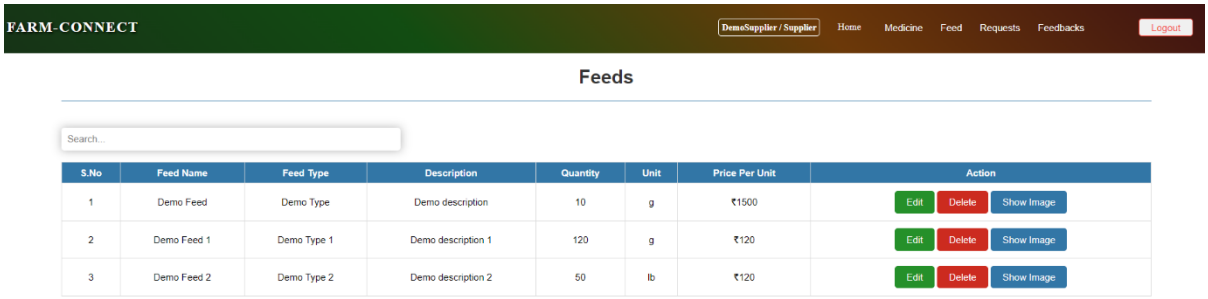


On Clicking the ‘Submit’ button a feed is added to the system and the supplier can add a new feed again. To move to other pages supplier can click any of the menus available in the navbar.

Supplier View Feed

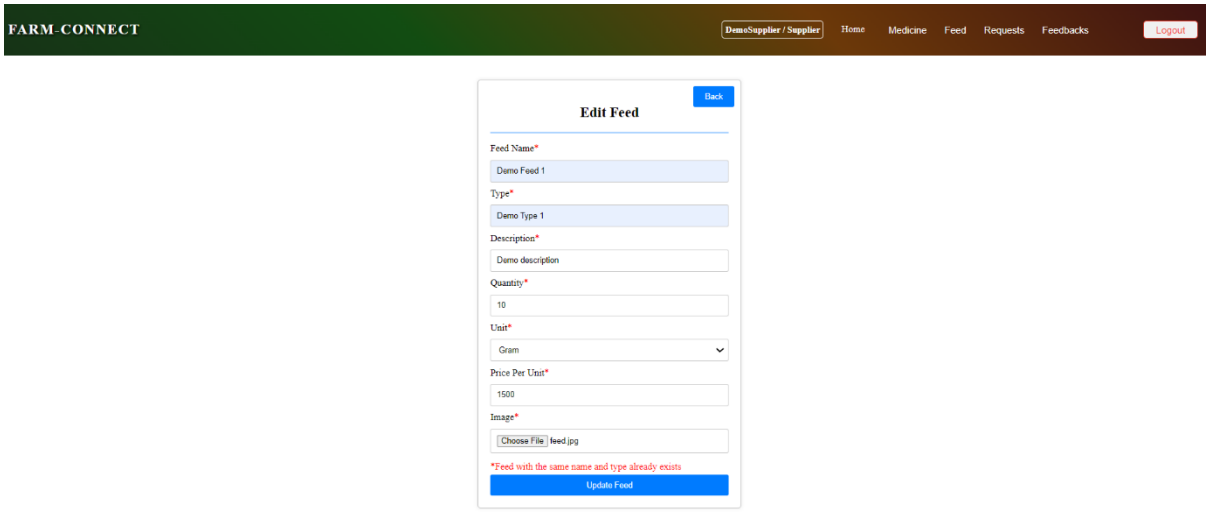
On hovering over the "**Feed**" item in the navbar, a submenu should appear with options to "**Add Feed**" and "**View Feed**".

Clicking on "**View Feed**" will navigate to the **viewfeed component**, which displays all Feed details in a table format with the heading "**Feeds**". Additionally, a search feature is provided to search based on Feed Name and Type.

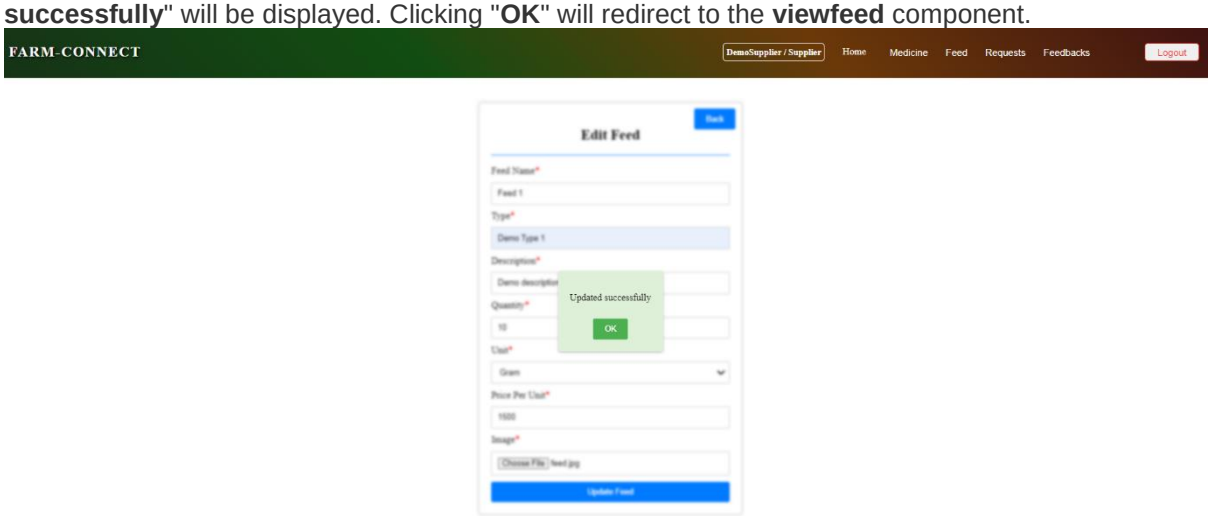


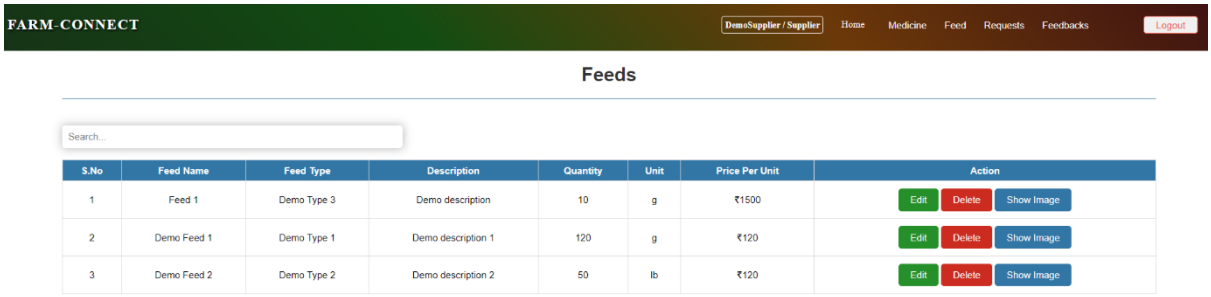
Clicking the **edit** button will navigate to the **supplierediteed component**, which displays an editing form with pre-populated feed data of the selected feed. This form also contains a "**Back**" button that will navigate to the **viewfeed component**. Validations are performed for all the form fields.

If the Supplier updates the feed name & type to match an existing one, an error indicating "**Feed with the same name and type already exists**" will be displayed.

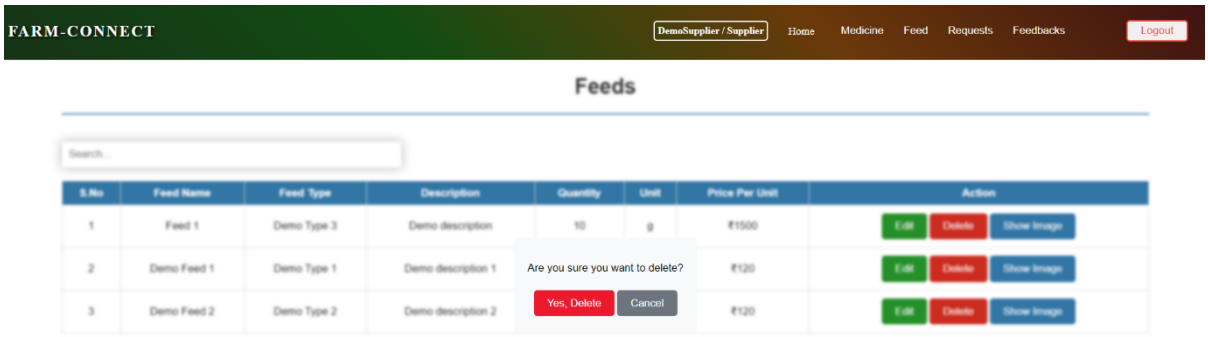


Upon clicking the "**Update Feed**" button, if the operation is successful, a popup saying "**Updated successfully**" will be displayed. Clicking "**OK**" will redirect to the **viewfeed** component.

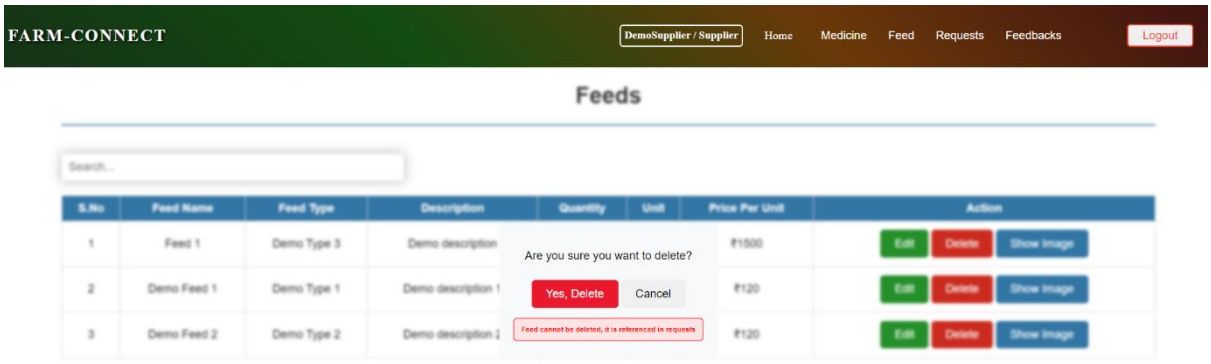




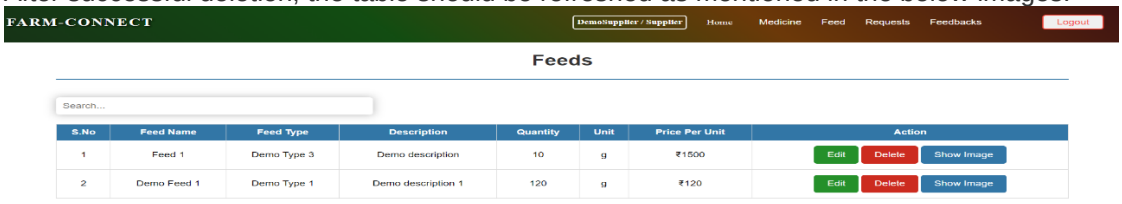
On clicking the "**Delete**" button, a pop-up should be displayed with a confirmatory message to delete the data.



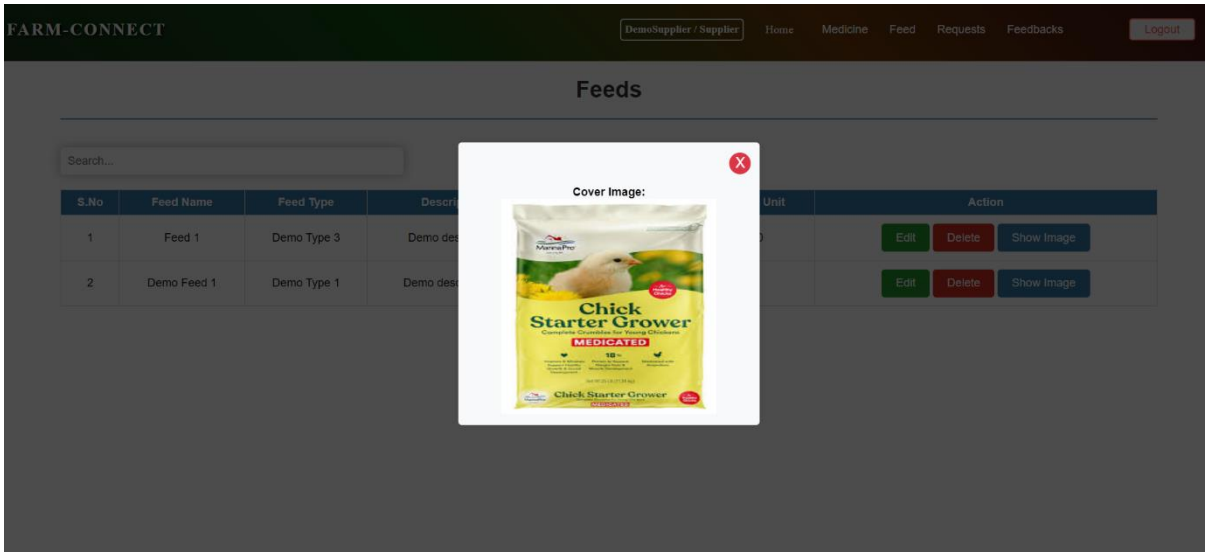
Once an Owner has requested for the feed, the supplier should not be able to delete the feed. Attempting to do so will display an error message stating **"Feed cannot be deleted, it is referenced in requests".**



After successful deletion, the table should be refreshed as mentioned in the below images.

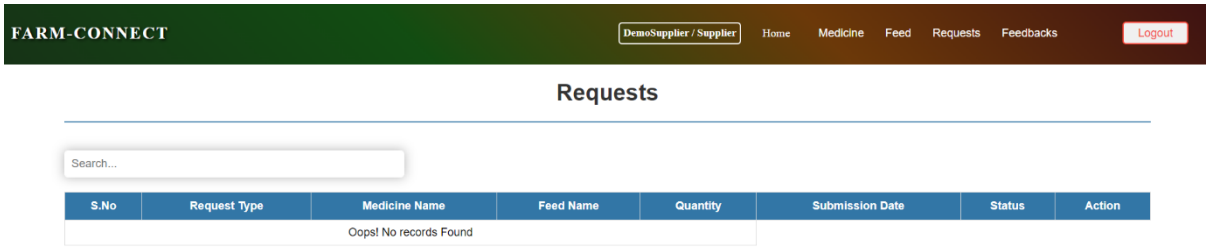


On clicking the **"Show Image"** button, the image of the feed should be displayed as a popup.

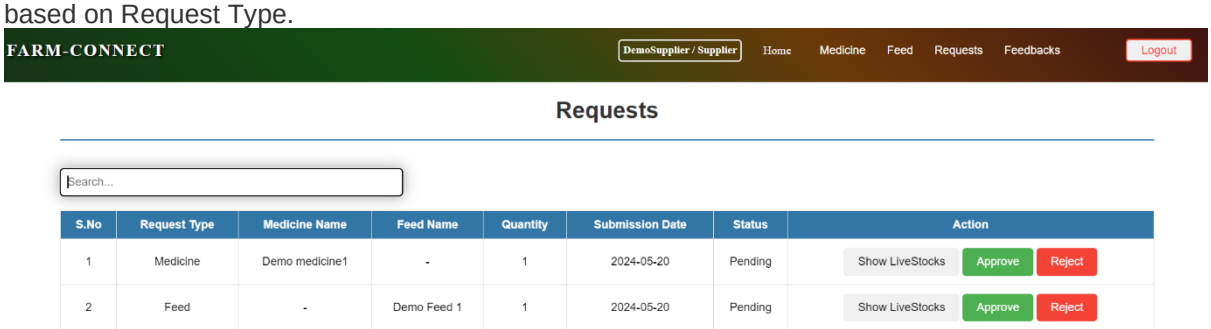


Supplier View Requests Send by the owners for Medicines & feeds:

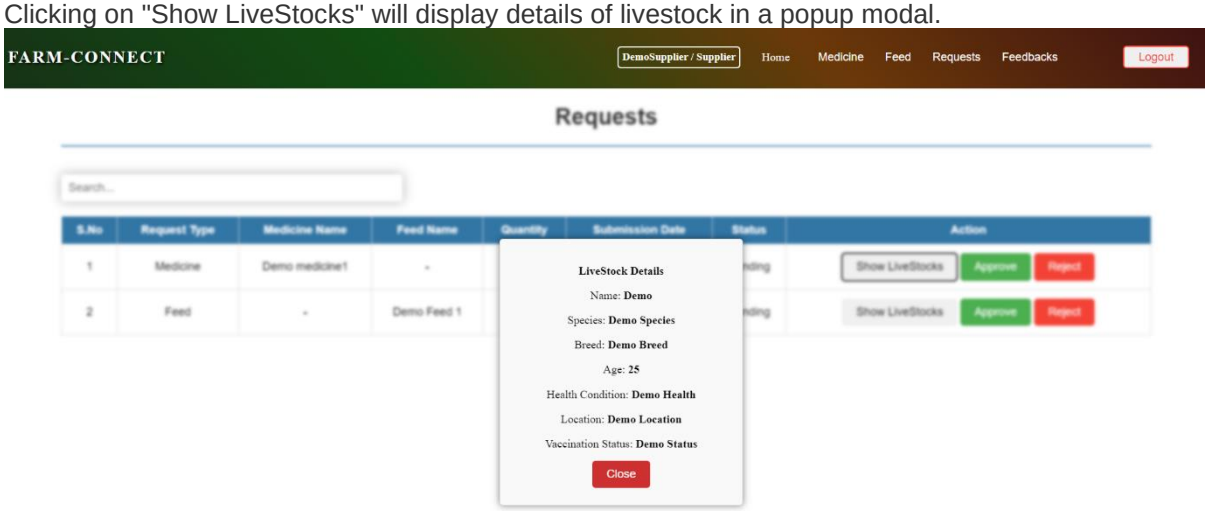
Clicking on "Requests" from the navbar will navigate to the **supplierrequests component**, which displays requests that match their medicines or feeds.
If no data is available, then "Oops! No records Found" should be displayed



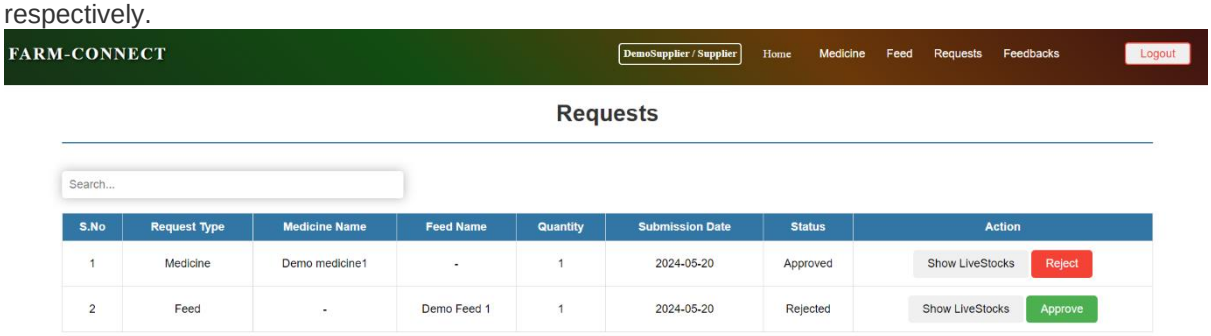
If data is available, the Requests requested by the owners that match the supplier's medicines & feeds will be displayed in table format. Additionally, features such as the Supplier being able to search based on Request Type.



Clicking on "Show LiveStocks" will display details of livestock in a popup modal.

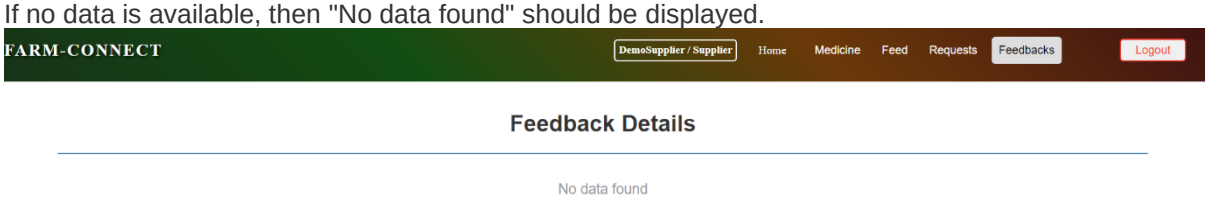


The Supplier can approve or reject requests by clicking on the "Approve" or "Reject" button, respectively.

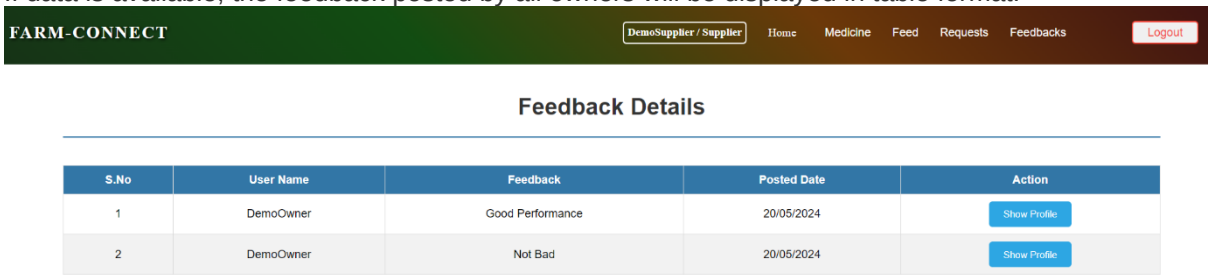


Supplier View Feedbacks:
Clicking on "Feedbacks" from the navbar will navigate to the **adminviewfeedback component**, which displays all feedbacks posted by all users.

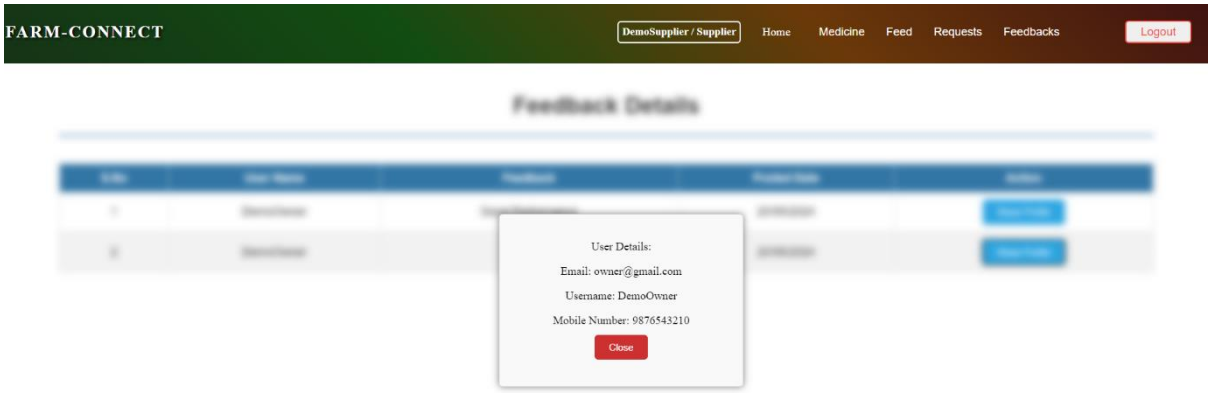
If no data is available, then "No data found" should be displayed.



If data is available, the feedback posted by all owners will be displayed in table format.

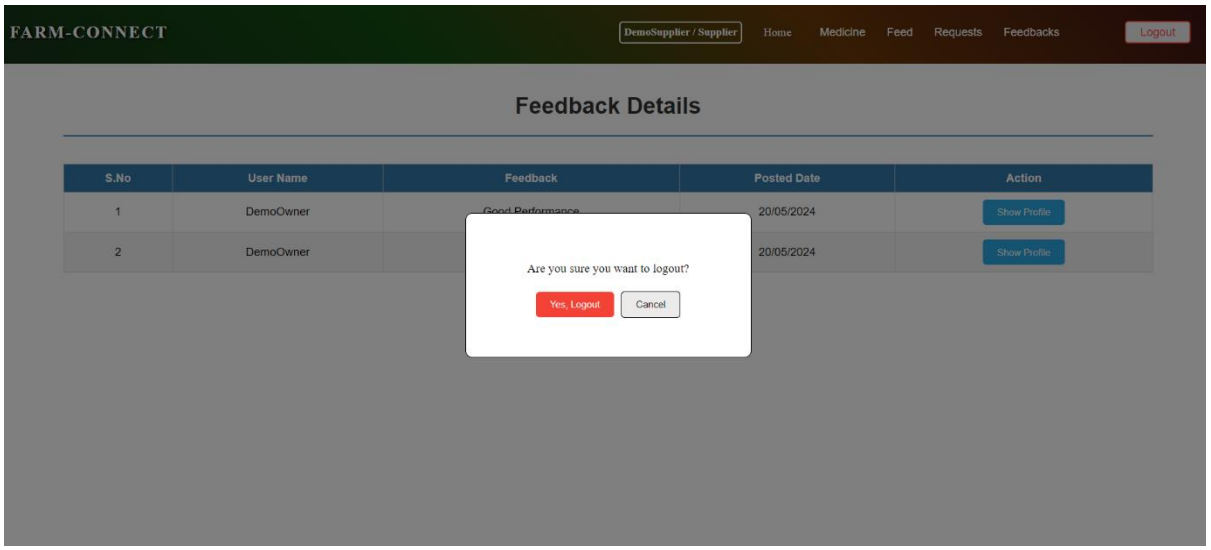


Clicking on "Show Profile" will display additional details about the user in pop-up modal.

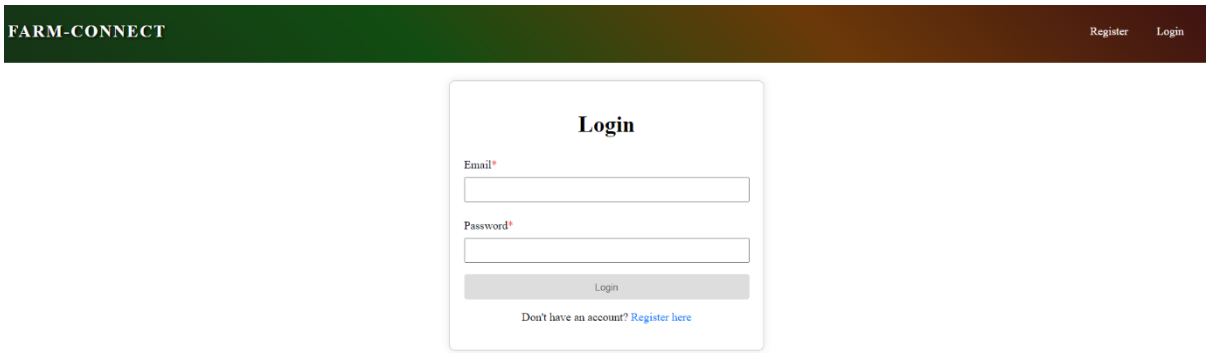


On clicking the "**Close**" button, will close the pop-up modal displayed.

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

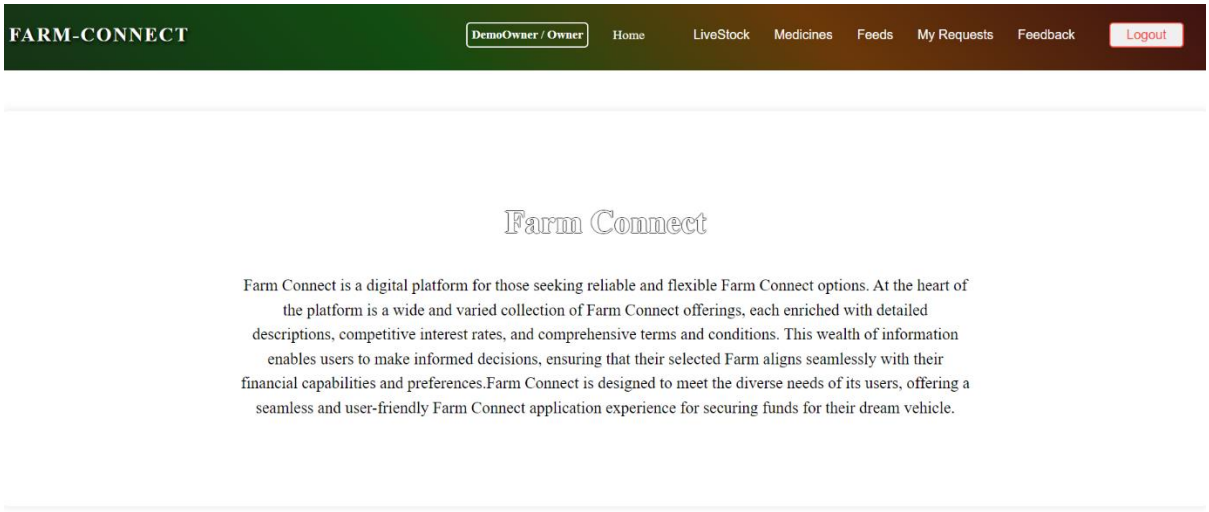


Clicking "Yes, Logout" will navigate to the login component.



Owner side (User Side):

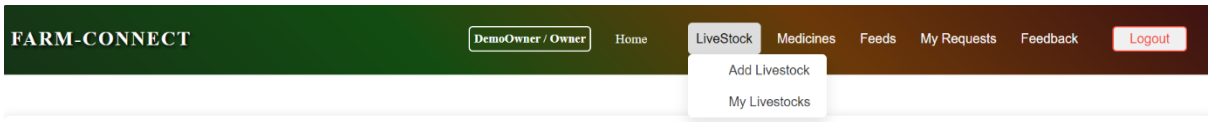
Home Component: This page is used to display the information about the Form Connect application. On clicking the ‘Home’ tab, owner can view the information about the application.



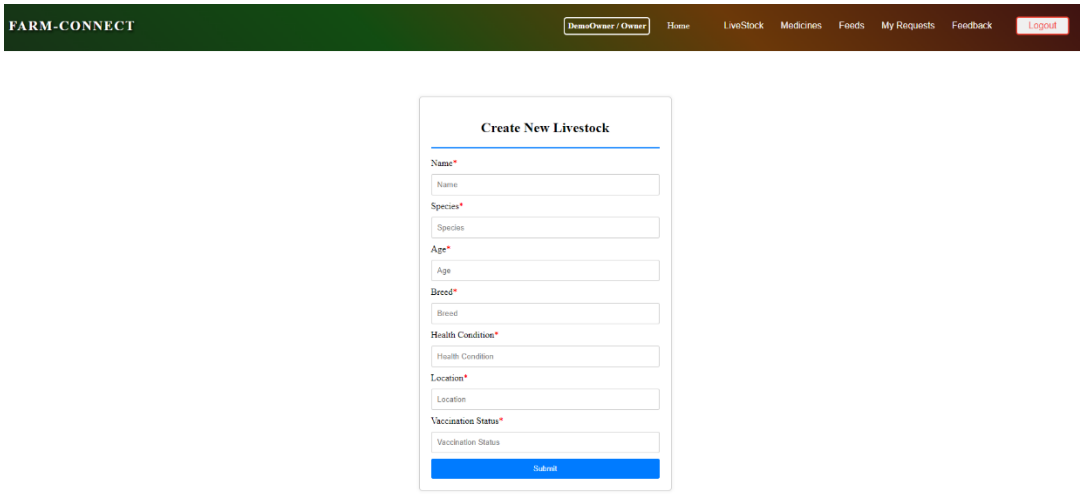
Upon successful login, if the user is an Supplier, the **(adminnav component)** will be displayed. If the user is a regular owner, the **(usernav component)** will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

Owner (User) can navigate to other pages by clicking on the menu available in the navigation bar (**usernav**).

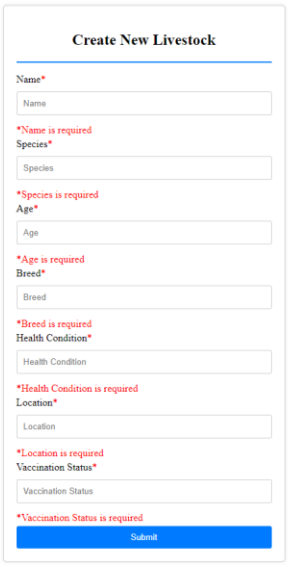
On hovering over the "**Livestock**" item in the navbar, a submenu should appear with options to "**Add Livestock**" and "**My Livestock**".



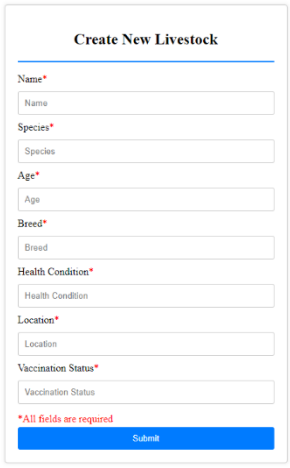
Clicking on "**Add Livestock**" will navigate to the **createlivestock component**, which displays a form with the heading "**Create New Livestock**". The Owner (user) should be able to add the **Livestock** details on this page, which is used to add a new Livestock to the system.

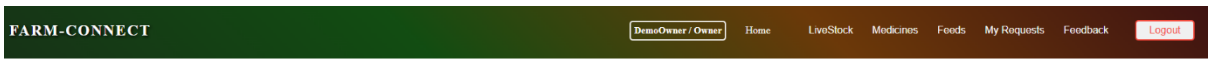


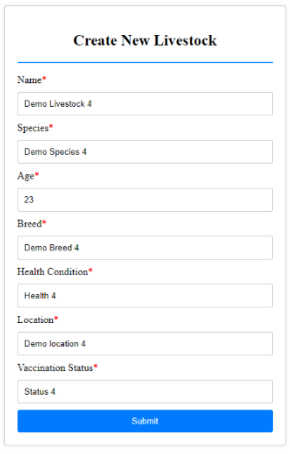
Perform validations for all the form fields.



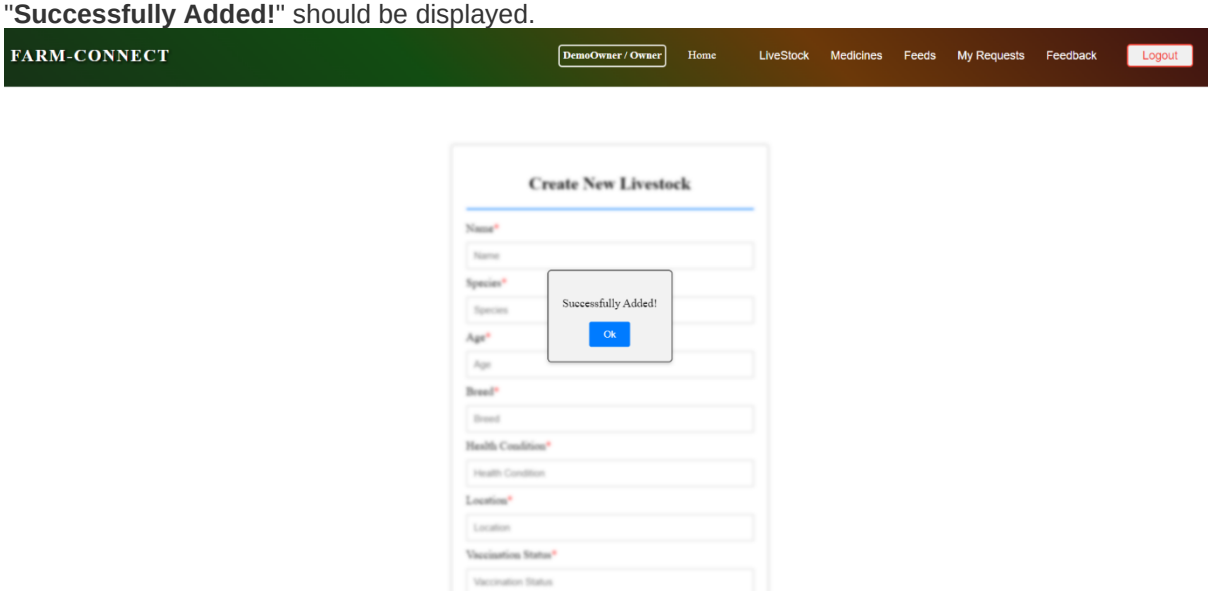
Clicking the "Submit" button with empty fields will display a validation message stating "All fields are required".







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



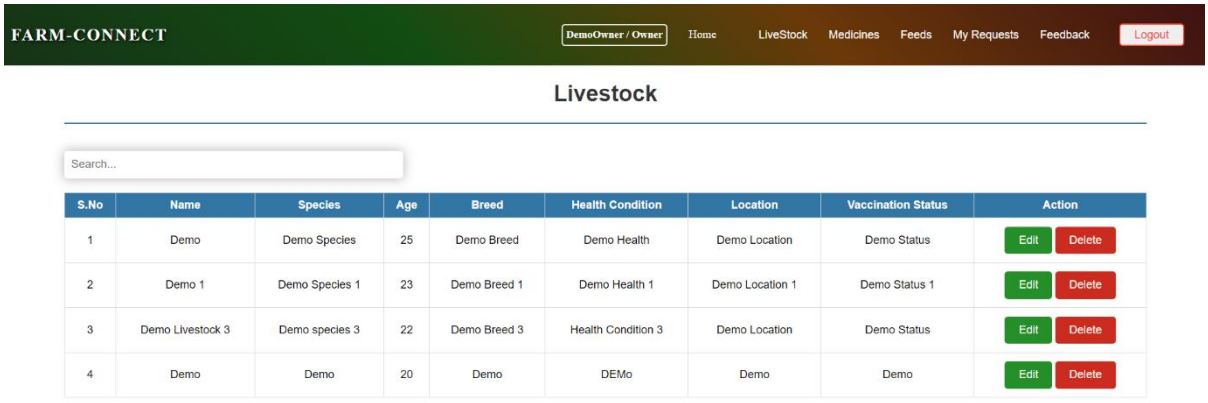
If the Owner (user) adds a new Livestock with an existing Name, Breed and Species, an error message stating "**Livestock with the same name, breed, and species already exists**" should be displayed.

On Clicking the 'Submit' button a livestock is added to the system and owner can add a new livestock again. To move to other pages owner can click any of the menus available in the navbar.

Owner View LiveStocks

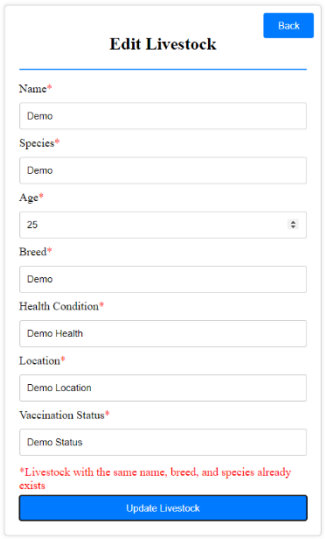
On hovering over the "**LiveStock**" item in the navbar, a submenu should appear with options to "**Add Livestock**" and "**My Livestock**".

Clicking on "**My Livestock**" will navigate to the **viewlivestock component**, which displays all Livestock of current owner details in a table format with the heading "**Livestock**". Additionally, a search feature is provided to search based on Name, Breed and Species.

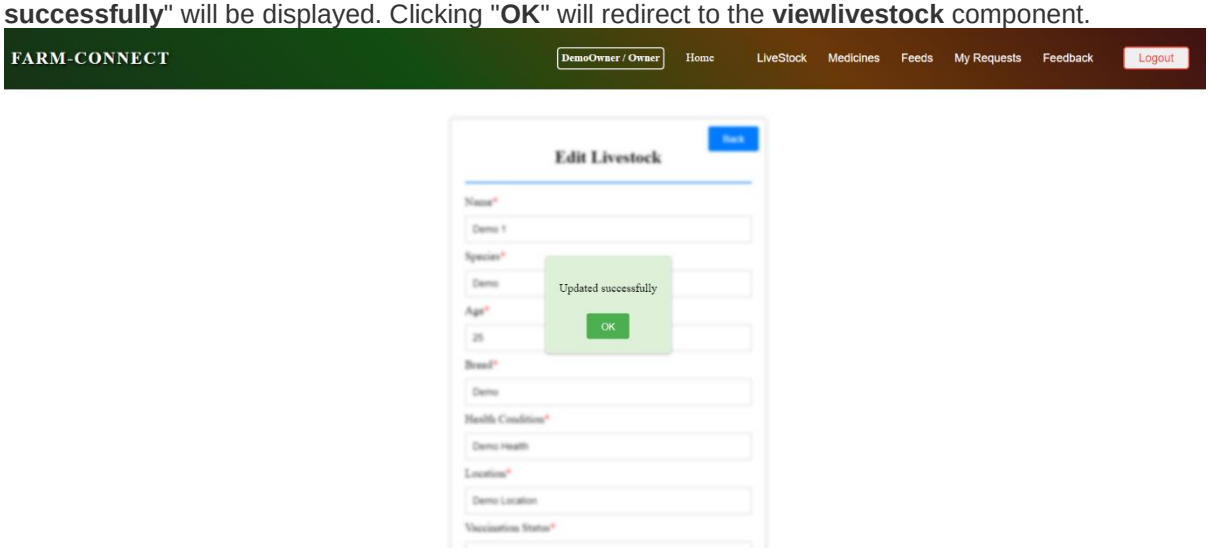


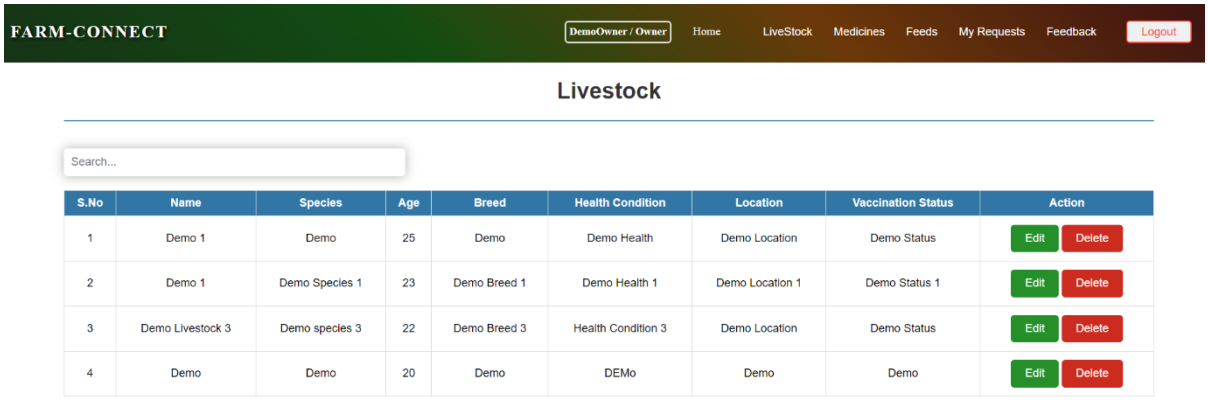
Clicking the edit button will navigate to the **ownereditlivestock component**, which displays an editing form with pre-populated livestock data of the selected livestock. This form also contains a "**Back**" button that will navigate to the **viewlivestock component**. Validations are performed for all the form fields.

If the Owner updates the Name, Breed and Species to match an existing one, an error indicating "**Livestock with the same name, breed, and species already exists**" will be displayed.

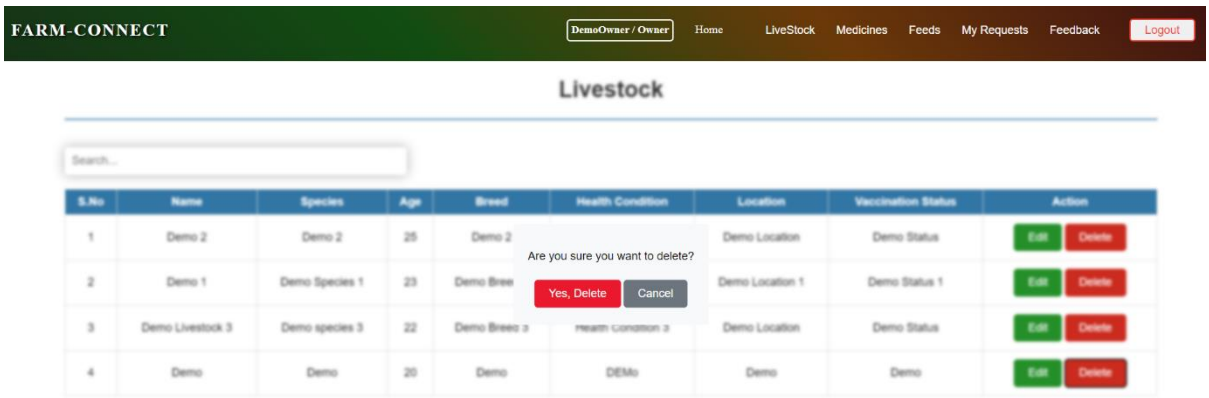


Upon clicking the "**Update Livestock**" button, if the operation is successful, a popup saying "**Updated successfully**" will be displayed. Clicking "**OK**" will redirect to the **viewlivestock** component.

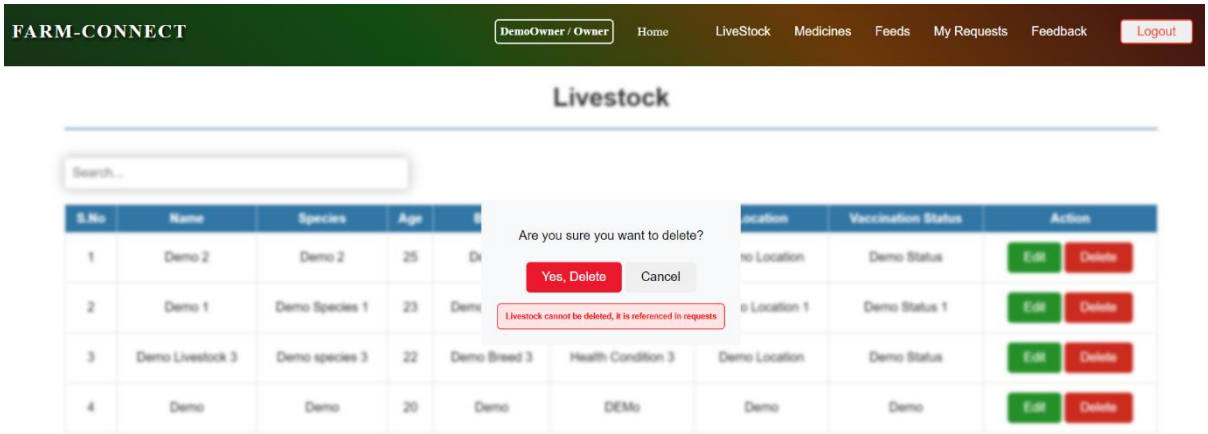




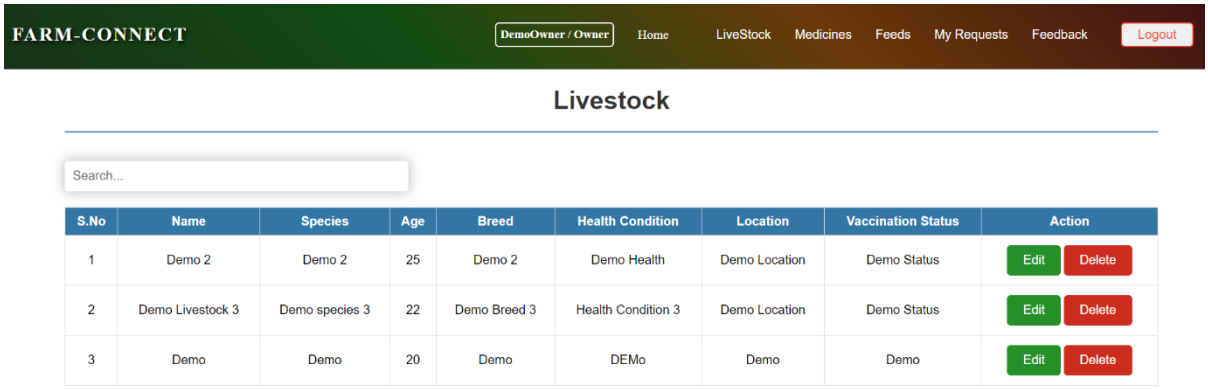
On clicking the "**Delete**" button, a pop-up should be displayed with a confirmatory message to delete the data.



Once an Owner has requested for the medicine/Feed for specific Livestock, the Owner should not be able to delete the livestock. Attempting to do so will display an error message stating **"Livestock cannot be deleted, it is referenced in requests".**



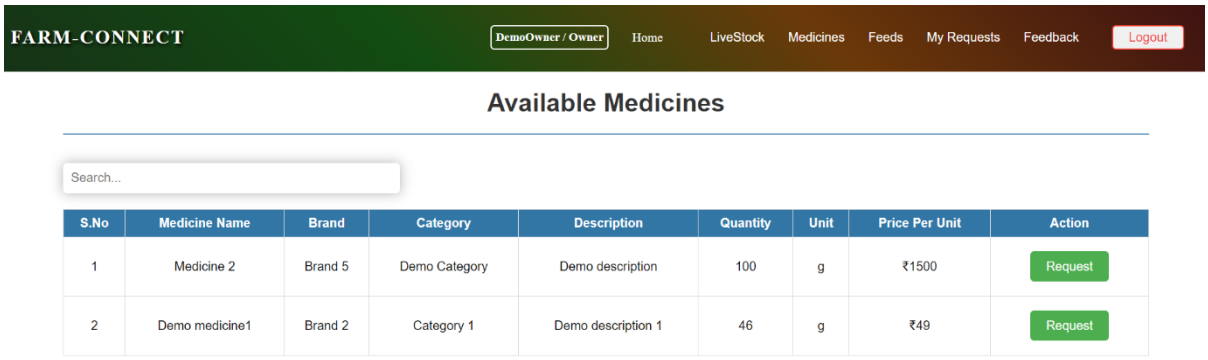
After successful deletion, the table should be refreshed as mentioned in the below images.



Owner View Medicines (user side):

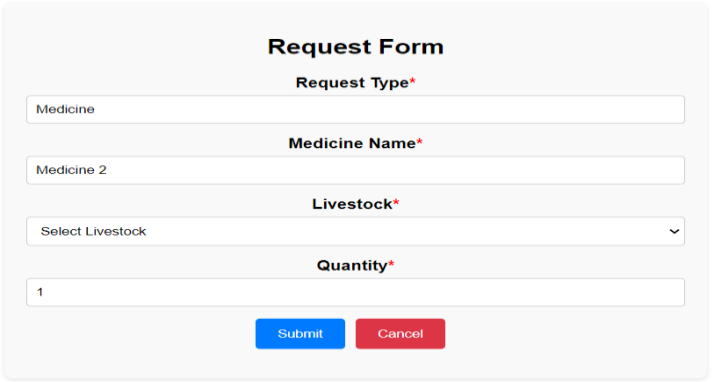
Clicking on "**Medicines**" from the navbar will navigate to the **ownerviewmedicine component**, which displays all medicines posted by Supplier.
If no data is available, then "Oops! No records Found" should be displayed.

Should display All medicines with action to request for medicines from supplier.

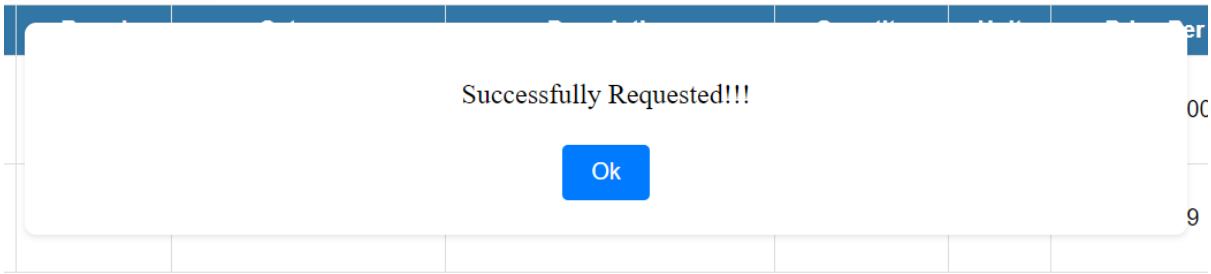


Clicking the "**Request**" button will navigate to the **requestform component**, which displays a form with the heading "**Request Form**". Additionally, it includes a "Cancel" button, which, when clicked, moves back to the **ownerviewmedicine component**

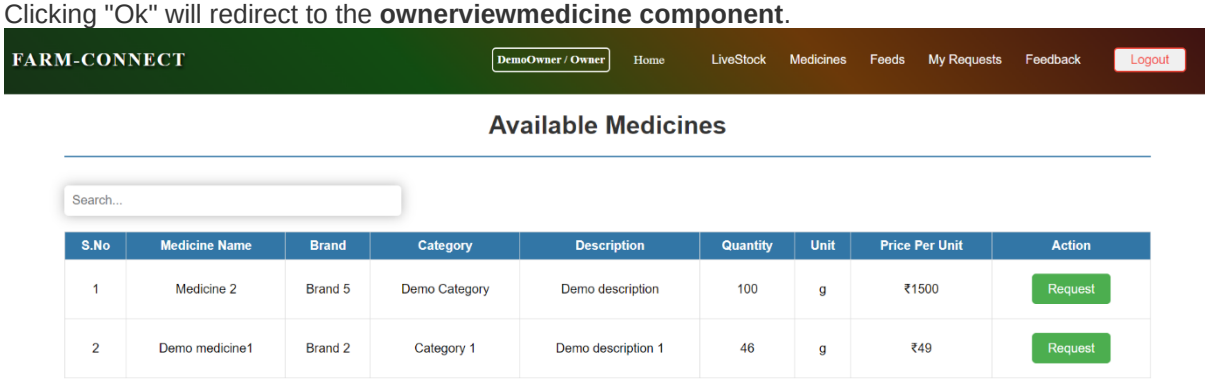
Here, Request Type & Medicine Name are read only mode & Livestock is dropdown to select one livestock from available live stocks.



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



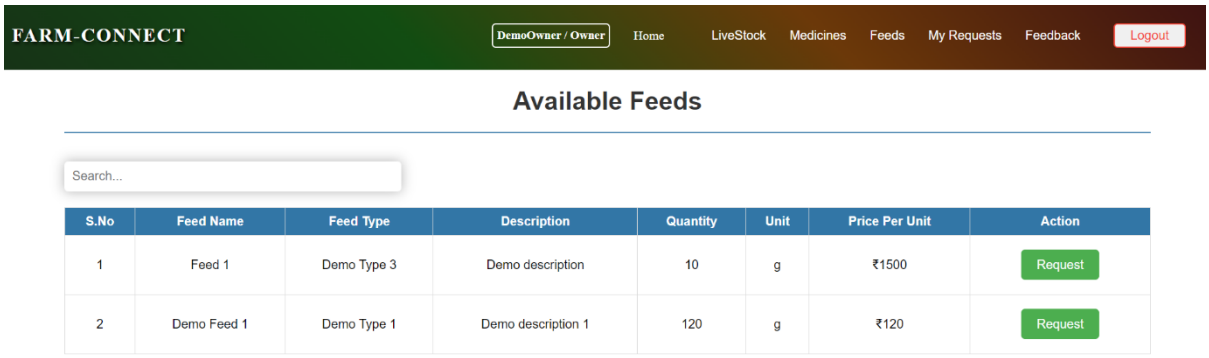
Clicking "Ok" will redirect to the **ownerviewmedicine component**.



Owner View Feeds (user side):

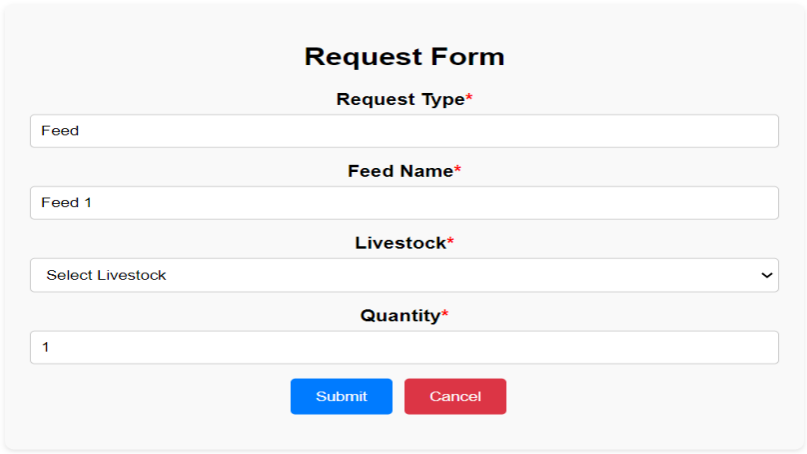
Clicking on "**Feeds**" from the navbar will navigate to the **ownerviewfeed component**, which displays all feeds posted by Supplier.
If no data is available, then "Oops! No records Found" should be displayed.

Should display All feeds with action to request for feeds from supplier.

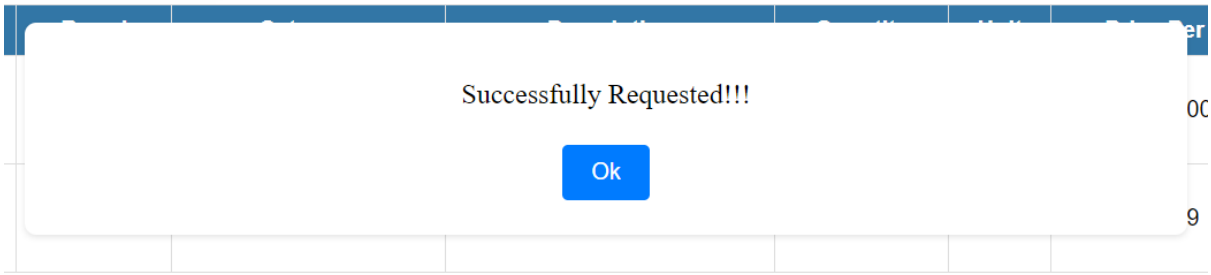


Clicking the "**Request**" button will navigate to the **requestform component**, which displays a form with the heading "**Request Form**". Additionally, it includes a "Cancel" button, which, when clicked, moves back to the **ownerviewfeed component**

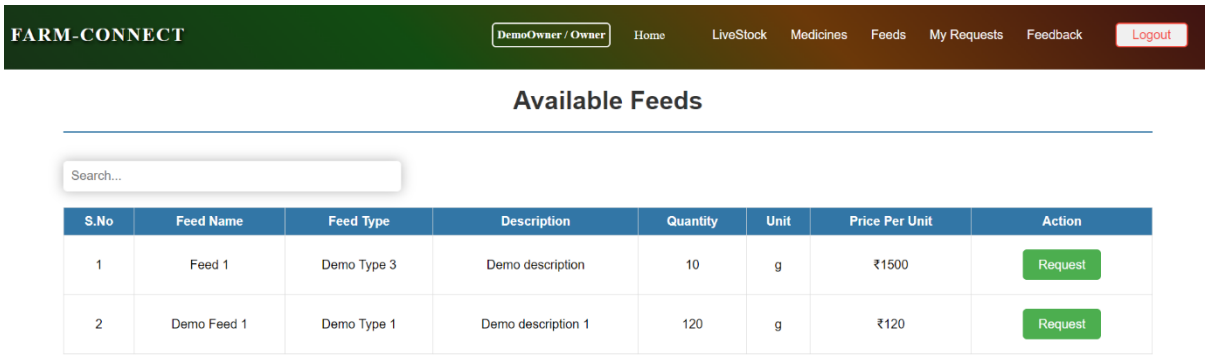
Here, Request Type & Feed Name are read only mode & Livestock is dropdown to select one livestock from available live stocks.



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

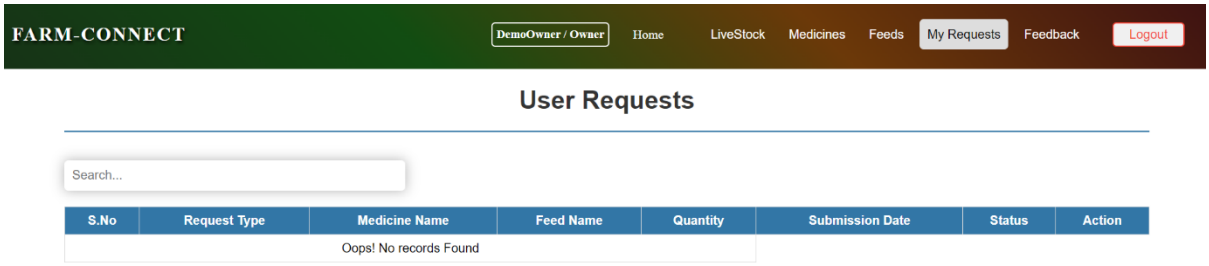


Clicking "Ok" will redirect to the **ownerviewfeed component**.

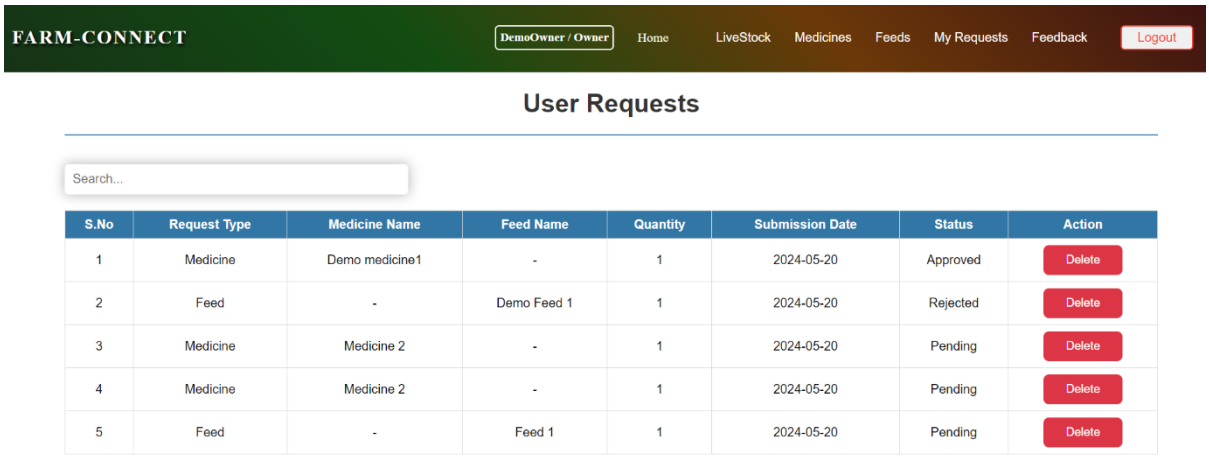


Owner - My Requests:

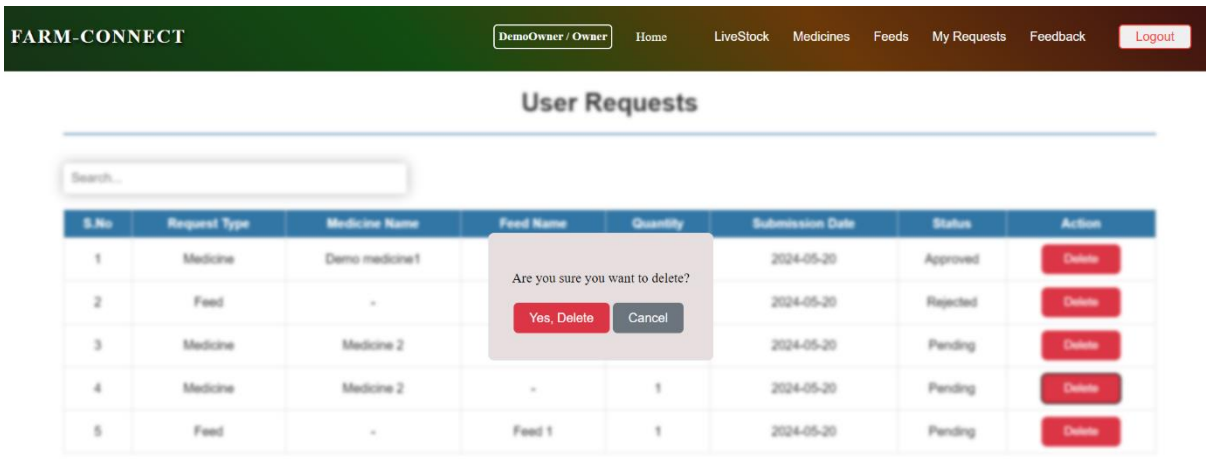
Clicking on "**My Requests**" from the navbar will navigate to the **ownerviewrequest component**, which displays the Owner Medicines & Feeds requests to Suppliers .
If no data is available, then "Oops! No records Found" should be displayed.



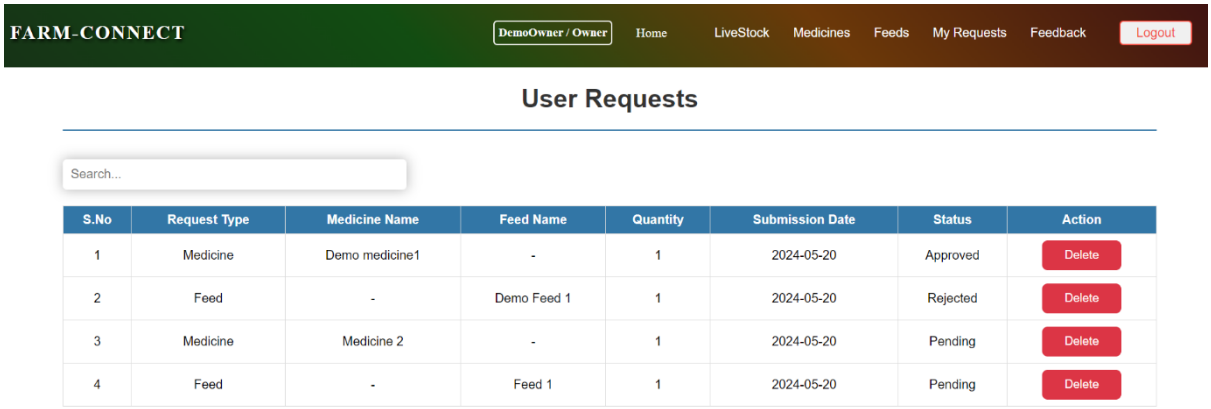
If data is available, the Requests requested will be displayed in table format. Additionally, features such as the owner being able to search based on Request Type.



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

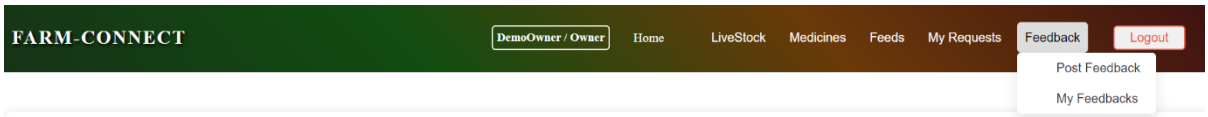


Clicking "**Yes, Delete**" will delete the Request, and the change will be automatically reflected.

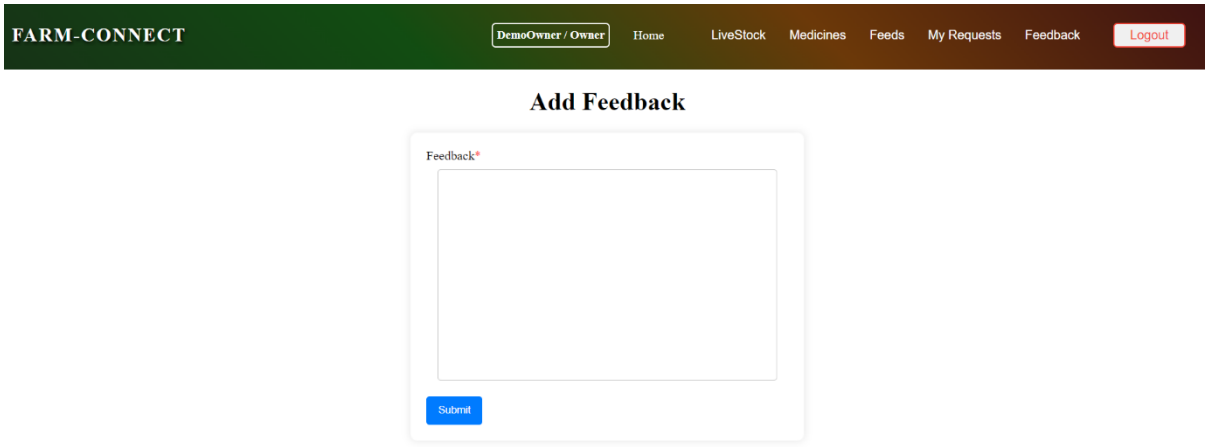


Owner Feedback:

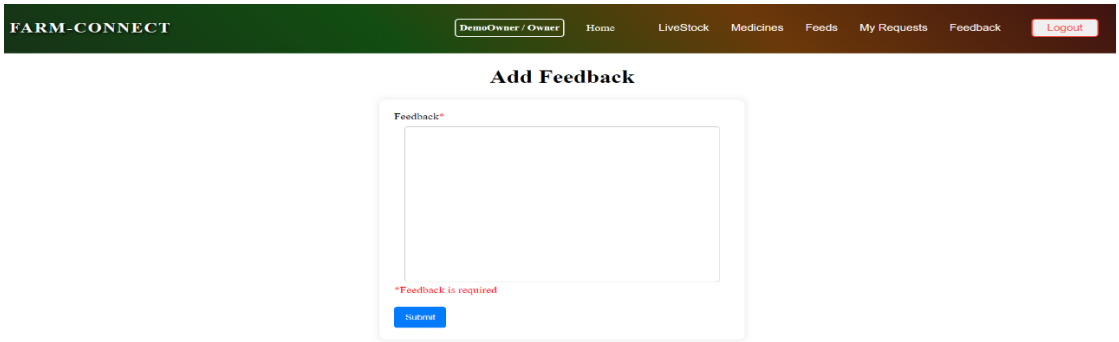
On hovering over the "**Feedback**" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".



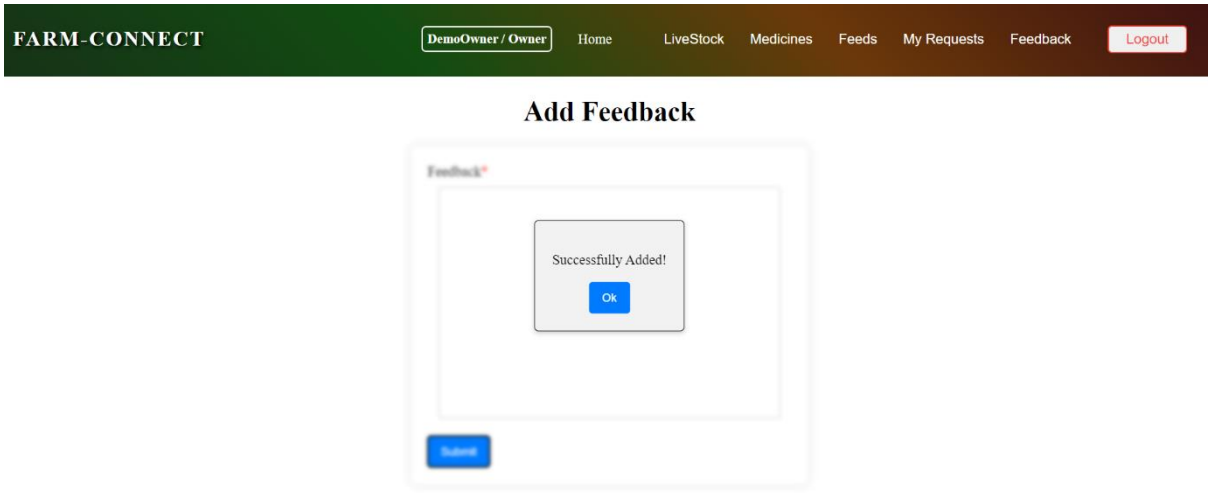
Clicking on **"Post Feedback"** will navigate to the **useraddfeedback component**, which displays the form to post feedback with heading as "**Add Feedback**"



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

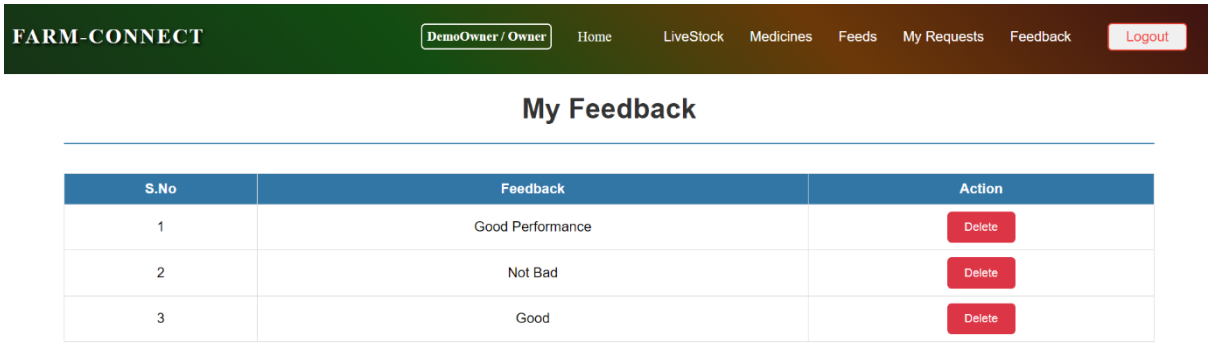


Upon clicking the "**Submit**" button, if the operation is successful, a popup message saying "**Successfully Added!**" should be displayed.

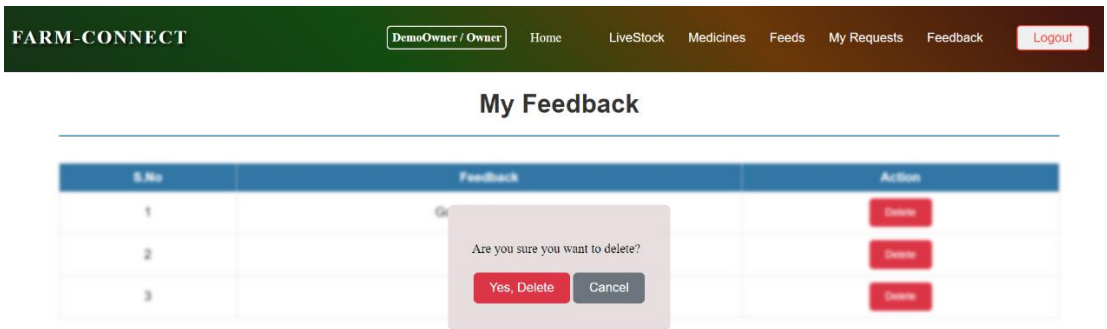


Clicking the "Ok" button will close the popup, and the same **useraddfeedback component** will be displayed.

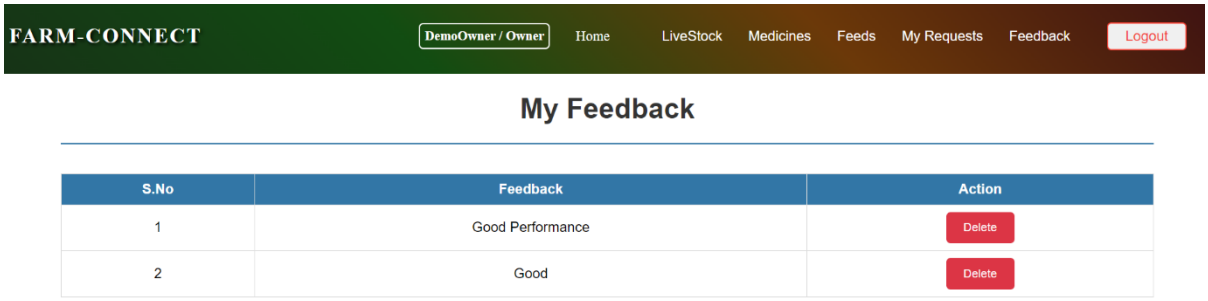
Owner view Feedback:
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "**Post Feedback**" and "**My Feedbacks**".
Clicking on "**My Feedbacks**" will navigate to the **userviewfeedback component**, which displays posted feedback with heading as "**My Feedback**"



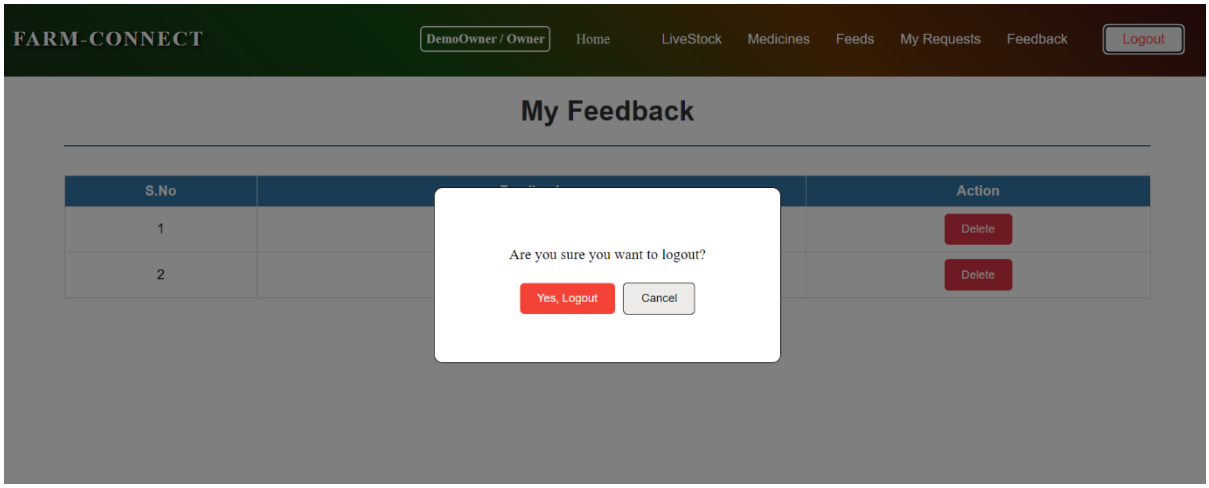
On clicking the "**Delete**" button, a pop-up should be displayed with confirmatory message to delete the data.



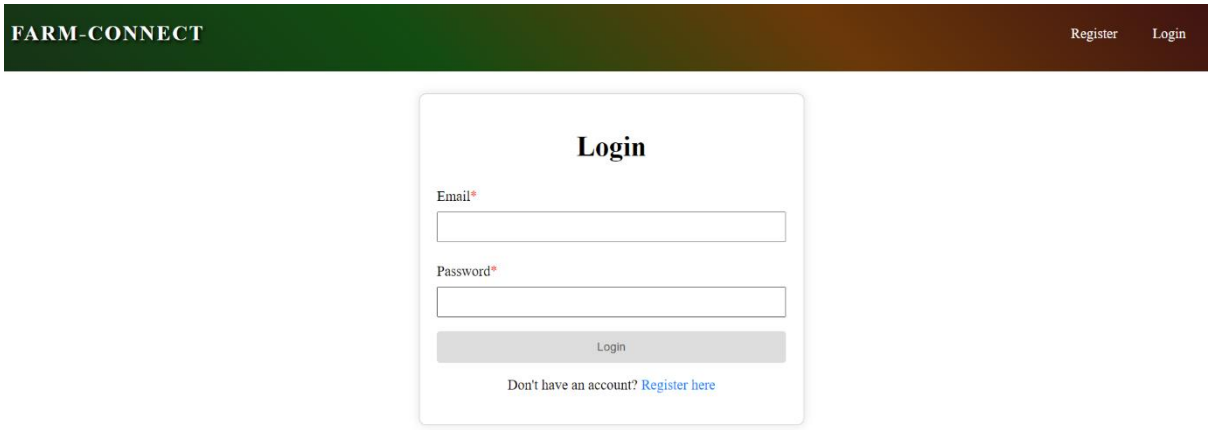
Clicking "**Yes, Delete**" will delete the feedback posted, and the change will be automatically reflected.



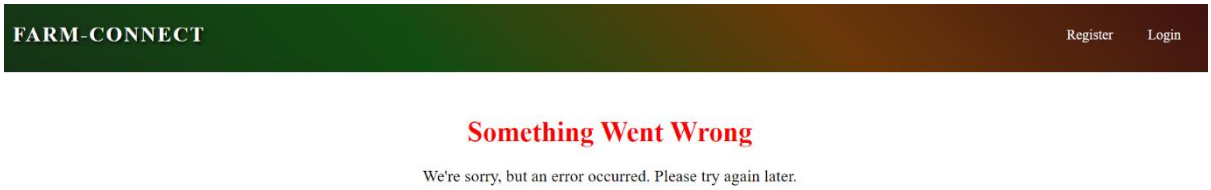
On clicking the "**Logout**" button, a pop-up should be displayed with confirmatory message to logout the user.



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



error component
This page displays an error message stating "**Something Went Wrong**".



Platform Prerequisites (Do's and Don'ts):

1. The angular app should run in port 8081.
2. The dotnet app should run in port 8080.
3. To incorporate .Net Security into the application, use JWT authentication within the project workspace.

Key points to remember:

1. The id (for frontend) and attributes(backend) mentioned in the SRS should not be modified at any cost. Failing to do may fail test cases.
2. Remember to check the screenshots provided with the SRS.
3. Strictly adhere to the proper project scaffolding (Folder structure), coding conventions, method definitions and return types.
4. Adhere strictly to the endpoints mentioned in API endpoints section.
5. Don't delete any files in a project environment.

HOW TO RUN THE PROJECT:

BACKEND:
Open the terminal and follow the commands below.

- **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

- **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**

>use DBName
>go

1> insert into TableName values(" "," ",...)
2> go

Note:

1. Please ensure that the application is running on port 8080 before clicking the "Run Test Case" button.
2. Database Name should be **appdb**
3. **Use the below sample connection string to connect the Ms SQL Server**

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

FRONTEND:

Open the terminal and follow the commands below.
Step 1:
• **Use "cd angularapp**" command to go inside the angularapp folder
• Install Node Modules - "**npm install**"

Step 2:
• Write the code inside src/app folder
• Create the necessary components
• To create Service: "**npx ng g s <service name>**"
• To create Component: "**npx ng g c <component name>**"

Step 3:
• Click the **Run Test Case** button to run the test cases

Note:
• Click PORT **8081** to view the result / output.
• If any error persists while running the app, delete the node modules and reinstall them.