**####\*\*\*This Description has only limited information. Kindly refer to the pdf document given in your learner course for more details. \*\*\*####**

**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.

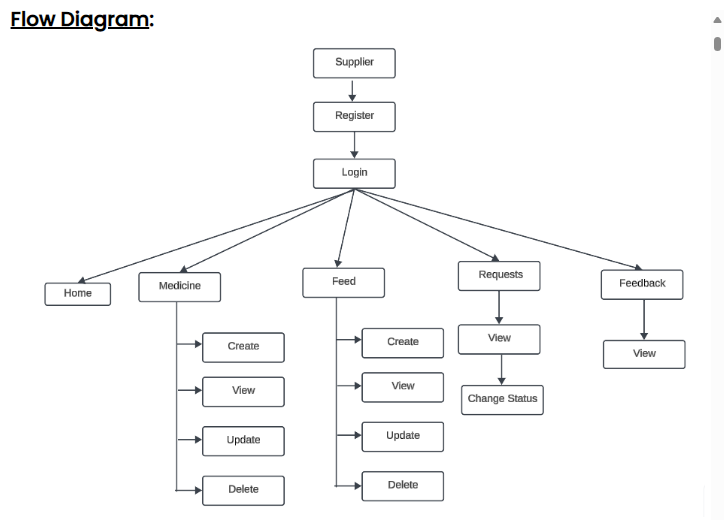
**Users of the System:**

1. Supplier

2.Owner

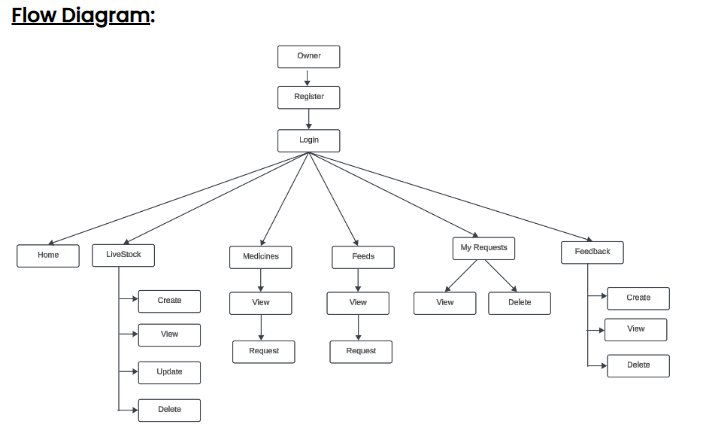
**Supplier Actions:**

**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

**Supplier:**

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.

**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.  **Supplier**: string - Represents the role of an Supplier user.

2.  **Owner**: string - Represents the role of a owner 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**:

**Frontend Requirements:**

**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.

**﻿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**