Implementing User-Based Note Isolation: Complete Guide

This guide walks you through implementing user-based note isolation where each user can only see and manage their own notes, while administrators can access all notes.

Overview

After completing this implementation, you will have:

- ✅ **User Isolation**: Regular users see only their own notes
- ✅ **Admin Privileges**: Admin users can see and manage all notes
- ✅ **Database Relations**: Foreign key relationship between Users and Notes
- ✅ **Security**: API endpoints enforce user ownership validation
- ✅ **Data Integrity**: Notes are automatically assigned to the current user

---

Part 1: Database Schema Changes

Step 1: Update the Note Model

File: `ListKeeper.ApiService/Models/Note.cs`

Add the foreign key relationship to the User model:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data;
using System.Numerics;
using ListKeeperWebApi.WebApi.Models; // Add this import

namespace ListKeeper.ApiService.Models
{
    [Table("Note")]
    public class Note
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        [StringLength(200)]
        public string Title { get; set; }

        [StringLength(500)]
        public string Content { get; set; }

        public DateTime DueDate { get; set; }
        public bool IsCompleted { get; set; }
        [StringLength(16)]
        public string Color { get; set; }

        /// <summary>
        /// Gets or sets the ID of the user who owns this note
        /// </summary>
        [Required]
        public int UserId { get; set; }

        /// <summary>
        /// Navigation property to the User who owns this note
        /// </summary>
        [ForeignKey("UserId")]
        public virtual User? User { get; set; }

        public Note()
        {
            Title = string.Empty;
            Content = string.Empty;
            Color = string.Empty;
        }
    }
}
```

💡 **Key Learning Points:**

- `[Required]` ensures every note must have an owner
- `[ForeignKey("UserId")]` creates the database relationship
- `virtual` enables Entity Framework lazy loading
- Navigation properties create object-oriented relationships

Step 2: Update the User Model

File: `ListKeeper.ApiService/Models/User.cs`

Add the import and navigation property:

```
using ListKeeperWebApi.WebApi.Models.Interfaces;
using ListKeeper.ApiService.Models; // Add this import
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

// ... existing code ...

/// <summary>
/// Navigation property to the notes owned by this user
/// </summary>
public virtual ICollection<Note>? Notes { get; set; }
```

💡 **Key Learning Points:**

- `ICollection<Note>` represents a one-to-many relationship
- One user can have many notes
- `virtual` enables Entity Framework lazy loading
- Navigation properties work in both directions

Step 3: Update the NoteViewModel

File: `ListKeeper.ApiService/Models/ViewModels/NoteViewModel.cs`

Add the UserId property:

```
public class NoteViewModel
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime DueDate { get; set; }
    public bool IsCompleted { get; set; }
    public string Color { get; set; }

    /// <summary>
    /// Gets or sets the ID of the user who owns this note
    /// </summary>
    public int UserId { get; set; }

    // ... constructor ...
}
```

Step 4: Configure Entity Framework Relationships

File: `ListKeeper.ApiService/Data/DatabaseContext.cs`

Update the `OnModelCreating` method:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Configure User-Note relationship
    modelBuilder.Entity<Note>()
        .HasOne(n => n.User)
        .WithMany(u => u.Notes)
        .HasForeignKey(n => n.UserId)
        .OnDelete(DeleteBehavior.Cascade); // When a user is deleted, delete their
notes

    // Create index on UserId for better query performance
    modelBuilder.Entity<Note>()
        .HasIndex(n => n.UserId)
```

```
            .HasDatabaseName("IX_Note_UserId");
    }
```

💡 **Key Learning Points:**

- `HasOne().WithMany()` defines the relationship cardinality
- `OnDelete(DeleteBehavior.Cascade)` handles related data cleanup
- Database indexes improve query performance for foreign keys
- Entity Framework uses fluent API for complex configurations

Step 5: Create and Run Migration

You will need to create and run this migration yourself:

```
# In ListKeeper.ApiService directory
dotnet ef migrations add AddUserIdToNotes
dotnet ef database update
```

Migration will:

- Add `UserId` column to Notes table
- Add foreign key constraint
- Create index on UserId
- **Note**: Existing notes will need UserId values assigned

Part 2: Current User Context

Step 6: Create CurrentUserHelper

File: `ListKeeper.ApiService/Helpers/CurrentUserHelper.cs`

Create a helper class to access current user information:

```
using System.Security.Claims;

namespace ListKeeper.ApiService.Helpers
{
    /// <summary>
    /// Helper class for working with current user context
    /// </summary>
    public class CurrentUserHelper
    {
        private readonly IHttpContextAccessor _httpContextAccessor;

        public CurrentUserHelper(IHttpContextAccessor httpContextAccessor)
        {
            _httpContextAccessor = httpContextAccessor ?? throw new
ArgumentNullException(nameof(httpContextAccessor));
```

```
        }

        /// <summary>
        /// Gets the current user's ID from the JWT token claims
        /// </summary>
        public int? GetCurrentUserId()
        {
            var userIdClaim =
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.NameIdentifier)?.Valu
e;
            if (string.IsNullOrEmpty(userIdClaim))
                return null;

            if (int.TryParse(userIdClaim, out var userId))
                return userId;

            return null;
        }

        /// <summary>
        /// Gets the current user's role from the JWT token claims
        /// </summary>
        public string? GetCurrentUserRole()
        {
            return
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.Role)?.Value;
        }

        /// <summary>
        /// Checks if the current user is an admin
        /// </summary>
        public bool IsCurrentUserAdmin()
        {
            var role = GetCurrentUserRole();
            return string.Equals(role, "Admin",
StringComparison.OrdinalIgnoreCase);
        }

        /// <summary>
        /// Gets the current user's username from the JWT token claims
        /// </summary>
        public string? GetCurrentUserName()
        {
            return
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.Name)?.Value;
        }
    }
}
```

Key Learning Points:

- JWT tokens contain user information as "claims"

- `ClaimTypes.NameIdentifier` is the standard claim for user ID
- `ClaimTypes.Role` contains the user's role
- Claims are extracted from the HTTP context of the current request

Step 7: Update JWT Token Generation

File: `ListKeeper.ApiService/EndPoints/UserEndpoints.cs`

Update the `GenerateJwtToken` method to include the user ID in standard claims:

```
var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(new[]
    {
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()), // Standard
claim for user ID
        new Claim("id", user.Id.ToString()), // Keep for backward compatibility
        new Claim(ClaimTypes.Name, user.Username ?? string.Empty),
        new Claim(ClaimTypes.Email, user.Email ?? string.Empty),
        new Claim(ClaimTypes.Role, user.Role ?? string.Empty)
    }),
    Expires = DateTime.UtcNow.AddDays(7),
    SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature)
};
```

💡 **Key Learning Points:**

- `ClaimTypes.NameIdentifier` is the standard way to store user ID
- Multiple claims can represent the same data for compatibility
- JWT tokens are signed to prevent tampering
- Claims are available throughout the request pipeline

Step 8: Register Helper in Dependency Injection

File: `ListKeeper.ApiService/Program.cs`

Add the import and register the helper:

```
using ListKeeper.ApiService.Helpers; // Add this import

// ... existing code ...

builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IUserService, UserService>();

builder.Services.AddScoped<INoteRepository, NoteRepository>();
builder.Services.AddScoped<INoteService, NoteService>();
```

```
// Register the helper for accessing current user information
builder.Services.AddScoped<CurrentUserHelper>();
```

Part 3: Repository Layer Updates

Step 9: Update Note Repository Interface

File: `ListKeeper.ApiService/Data/INoteRepository.cs`

Add user-specific methods:

```
public interface INoteRepository
{
    Task<Note> AddAsync(Note note);
    Task<bool> Delete(int id);
    Task<bool> Delete(Note note);
    Task<IEnumerable<Note>> GetAllAsync();
    Task<IEnumerable<Note>> GetAllAsync(int userId); // New method
    Task<IEnumerable<Note>> GetBySearchCriteriaAsync(SearchCriteria
searchCriteria);
    Task<IEnumerable<Note>> GetBySearchCriteriaAsync(SearchCriteria
searchCriteria, int userId); // New method
    Task<Note?> GetByIdAsync(int id);
    Task<Note?> GetByIdAsync(int id, int userId); // New method
    Task<Note> Update(Note note);
}
```

Step 10: Implement User-Specific Repository Methods

File: `ListKeeper.ApiService/Data/NoteRepository.cs`

Add the new method implementations:

```
/// <summary>
/// Finds a note by their primary key (ID) that belongs to a specific user.
/// </summary>
public async Task<Note?> GetByIdAsync(int id, int userId)
{
    _logger.LogInformation("Attempting to find note by ID: {NoteId} for user:
{UserId}", id, userId);
    try
    {
        return await _context.Notes
            .Where(n => n.Id == id && n.UserId == userId)
            .FirstOrDefaultAsync();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while getting note by ID: {NoteId}
for user: {UserId}", id, userId);
        throw;
    }
}

/// <summary>
/// Retrieves a list of all notes from the database for a specific user.
/// </summary>
public async Task<IEnumerable<Note>> GetAllAsync(int userId)
{
    _logger.LogInformation("Attempting to get all notes for user: {UserId}",
userId);
    try
    {
        return await _context.Notes
            .Where(n => n.UserId == userId)
            .ToListAsync();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while getting all notes for user:
{UserId}", userId);
        throw;
    }
}

/// <summary>
/// Retrieves notes based on search criteria for a specific user.
/// </summary>
public async Task<IEnumerable<Note>> GetBySearchCriteriaAsync(SearchCriteria
searchCriteria, int userId)
{
    _logger.LogInformation("Attempting to get notes by search criteria for user:
{UserId}", userId);
    try
    {
        var query = _context.Notes.Where(n => n.UserId == userId); // Start with
user filter
        var today = DateTime.Today;

        // Add search text filter if provided
        if (!string.IsNullOrWhiteSpace(searchCriteria.SearchText))
        {
            var searchText = searchCriteria.SearchText.ToLowerInvariant();
            query = query.Where(n =>
                n.Title.ToLower().Contains(searchText) ||
                n.Content.ToLower().Contains(searchText));
        }

        // Add completion status filter if specified
        if (searchCriteria.ShowOnlyCompleted.HasValue)
        {
            query = query.Where(n => n.IsCompleted ==
searchCriteria.ShowOnlyCompleted.Value);
        }

        // Add status filters if not "All"
        if (searchCriteria.Statuses.Length > 0 &&
!searchCriteria.Statuses.Contains(0))
        {
            var hasUpcoming = searchCriteria.Statuses.Contains(1);
            var hasPastDue = searchCriteria.Statuses.Contains(2);
            var hasCompleted = searchCriteria.Statuses.Contains(3);

            query = query.Where(n =>
                (hasUpcoming && n.DueDate.Date > today && !n.IsCompleted) ||
                (hasPastDue && n.DueDate.Date < today && !n.IsCompleted) ||
                (hasCompleted && n.IsCompleted));
        }

        return await query.ToListAsync();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while getting notes by search
criteria for user: {UserId}", userId);
        throw;
    }
}
```

💡 **Key Learning Points:**

- User filtering happens at the database level for security
- All queries start with `.Where(n => n.UserId == userId)`
- Entity Framework translates LINQ to SQL efficiently
- Database-level filtering prevents data leakage

Part 4: Service Layer Updates

Step 11: Update Note Service Interface

File: `ListKeeper.ApiService/Services/INoteService.cs`

Add user-specific method signatures:

```
public interface INoteService
{
    Task<NoteViewModel?> CreateNoteAsync(NoteViewModel createNoteVm);
    Task<bool> DeleteNoteAsync(int id);
    Task<bool> DeleteNoteAsync(NoteViewModel noteVm);
    Task<IEnumerable<NoteViewModel>> GetAllNotesAsync();
    Task<IEnumerable<NoteViewModel>> GetAllNotesAsync(int userId); // New method
    Task<IEnumerable<NoteViewModel>>
GetAllNotesBySearchCriteriaAsync(SearchCriteriaViewModel searchCriteria);
    Task<IEnumerable<NoteViewModel>>
GetAllNotesBySearchCriteriaAsync(SearchCriteriaViewModel searchCriteria, int
userId); // New method
    Task<NoteViewModel?> GetNoteByIdAsync(int id);
    Task<NoteViewModel?> GetNoteByIdAsync(int id, int userId); // New method
    Task<NoteViewModel?> UpdateNoteAsync(NoteViewModel noteVm);
}
```

Step 12: Implement User-Specific Service Methods

File: `ListKeeper.ApiService/Services/NoteService.cs`

Update existing methods and add new ones:

```
/// <summary>
/// Creates a new note in the system.
/// </summary>
public async Task<NoteViewModel?> CreateNoteAsync(NoteViewModel createNoteVm)
{
    if (createNoteVm == null) return null;

    var note = new Note
    {
        Title = createNoteVm.Title,
        DueDate = createNoteVm.DueDate,
        Color = createNoteVm.Color,
        Content = createNoteVm.Content,
        Id = createNoteVm.Id,
        IsCompleted = createNoteVm.IsCompleted,
        UserId = createNoteVm.UserId // Assign the note to the user
    };

    var createdNote = await _repo.AddAsync(note);
    return createdNote?.ToViewModel();
}

/// <summary>
/// Retrieves all notes from the system for a specific user.
/// </summary>
public async Task<IEnumerable<NoteViewModel>> GetAllNotesAsync(int userId)
{
    var notes = await _repo.GetAllAsync(userId);
    return notes?.Select(u => u.ToViewModel()).Where(vm => vm !=
null).Cast<NoteViewModel>() ?? Enumerable.Empty<NoteViewModel>();
}

/// <summary>
/// Retrieves a single note by their ID for a specific user.
/// </summary>
public async Task<NoteViewModel?> GetNoteByIdAsync(int id, int userId)
```

```
{
    var note = await _repo.GetByIdAsync(id, userId);
    return note?.ToViewModel();
}

/// <summary>
/// Retrieves notes based on search criteria for a specific user.
/// </summary>
public async Task<IEnumerable<NoteViewModel>>
GetAllNotesBySearchCriteriaAsync(SearchCriteriaViewModel searchCriteria, int
userId)
{
    // If "All" status is selected (0) or no statuses provided, and no other
filters, use the simple GetAll
    if ((searchCriteria.Statuses.Contains(0) || searchCriteria.Statuses.Length ==
0) &&
        string.IsNullOrWhiteSpace(searchCriteria.SearchText) &&
        !searchCriteria.ShowOnlyCompleted.HasValue)
    {
        return await GetAllNotesAsync(userId);
    }

    var domainSearchCriteria = searchCriteria.ToDomain();
    var notes = await _repo.GetBySearchCriteriaAsync(domainSearchCriteria,
userId);
    return notes?.Select(n => n.ToViewModel()).Where(vm => vm !=
null).Cast<NoteViewModel>() ?? Enumerable.Empty<NoteViewModel>();
}
```

Step 13: Update Note Mapping Extensions

File: `ListKeeper.ApiService/Models/Extensions/NoteMappingExtensions.cs`

Include UserId in the mapping methods:

```
public static NoteViewModel? ToViewModel(this Note? note)
{
    if (note == null) return null;

    return new NoteViewModel
    {
        Id = note.Id,
        Color = note.Color,
        Content = note.Content,
        DueDate = note.DueDate,
        IsCompleted = note.IsCompleted,
        Title = note.Title,
        UserId = note.UserId // Include user ownership
    };
}
```

```
public static Note? ToDomain(this NoteViewModel? viewModel)
{
    if (viewModel == null) return null;

    return new Note
    {
        Id = viewModel.Id,
        Color = viewModel.Color,
        Content = viewModel.Content,
        DueDate = viewModel.DueDate,
        IsCompleted = viewModel.IsCompleted,
        Title = viewModel.Title,
        UserId = viewModel.UserId // Include user ownership
    };
}
```

Part 5: API Endpoint Updates

Step 14: Update Note Endpoints with User Isolation

File: `ListKeeper.ApiService/EndPoints/NoteEndpoints.cs`

Add the import and update all endpoint methods:

```
using ListKeeper.ApiService.Helpers; // Add this import

// Update GetAllNotes method
private static async Task<IResult> GetAllNotes(
    [FromServices] INoteService noteService,
    [FromServices] CurrentUserHelper currentUserHelper,
    [FromServices] ILoggerFactory loggerFactory)
{
    var logger = loggerFactory.CreateLogger("Notes");
    try
    {
        logger.LogInformation("Getting all notes");

        // Check if current user is admin
        if (currentUserHelper.IsCurrentUserAdmin())
        {
            // Admin can see all notes
            var allNotes = await noteService.GetAllNotesAsync();
            return Results.Ok(new { notes = allNotes });
        }
        else
        {
            // Regular users only see their own notes
            var userId = currentUserHelper.GetCurrentUserId();
            if (!userId.HasValue)
            {
                return Results.Unauthorized();
            }

            var userNotes = await noteService.GetAllNotesAsync(userId.Value);
            return Results.Ok(new { notes = userNotes });
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Error retrieving all notes");
        return Results.Problem("An error occurred while retrieving notes",
statusCode: (int)HttpStatusCode.InternalServerError);
    }
}

// Update CreateNote method
private static async Task<IResult> CreateNote(
    [FromBody] NoteViewModel noteViewModel,
    [FromServices] INoteService noteService,
    [FromServices] CurrentUserHelper currentUserHelper,
    [FromServices] ILoggerFactory loggerFactory)
{
    var logger = loggerFactory.CreateLogger("Notes");
    try
    {
        logger.LogInformation("Creating new note");

        if (noteViewModel == null)
        {
            return Results.BadRequest("Note data is required");
        }

        // Get current user ID and assign it to the note
        var userId = currentUserHelper.GetCurrentUserId();
        if (!userId.HasValue)
        {
            return Results.Unauthorized();
        }

        // Always assign the note to the current user (security measure)
        noteViewModel.UserId = userId.Value;

        var createdNote = await noteService.CreateNoteAsync(noteViewModel);

        if (createdNote == null)
        {
            return Results.Problem("Failed to create note", statusCode:
(int)HttpStatusCode.InternalServerError);
        }

        return Results.Created($"/api/notes/{createdNote.Id}", createdNote);
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Error creating note");
        return Results.Problem("An error occurred while creating the note",
statusCode: (int)HttpStatusCode.InternalServerError);
    }
}
```

Apply similar patterns to UpdateNote, DeleteNote, GetNoteById, and GetAllNotesBySearchCriteria methods.

💡 **Key Learning Points:**

- Security is enforced at the API level
- Admin users get different behavior than regular users
- User ID is always extracted from the JWT token, never trusted from the client
- Unauthorized requests return 401 status codes

Part 6: Testing the Implementation

Step 15: Test User Isolation

Test Scenarios:

1. **Regular User Operations:**

```
# Sign up a regular user
POST /api/users/signup
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john@example.com",
  "username": "johndoe",
  "password": "password123",
  "confirmPassword": "password123",
  "agreeToTerms": true
}

# Create notes (should be assigned to john automatically)
POST /api/notes
{
  "title": "John's Note",
  "content": "This is John's private note",
  "dueDate": "2025-08-01",
  "color": "#FFF3CD",
  "isCompleted": false
}

# Get all notes (should only see john's notes)
GET /api/notes
```

2. **Admin User Operations:**

```
# Create an admin user (update user role to "Admin" in database)
# Login as admin
POST /api/users/authenticate
{
  "email": "admin@example.com",
  "password": "adminpassword"
}

# Get all notes (should see notes from all users)
GET /api/notes
```

3. **Cross-User Access Test:**

```
# Try to access another user's note by ID (should return 404)
GET /api/notes/[other_user_note_id]
```

Step 16: Database Verification

Check the database to verify:

```sql
-- Verify note ownership
SELECT n.Id, n.Title, n.UserId, u.Email
FROM Note n
JOIN Users u ON n.UserId = u.Id;

-- Verify foreign key constraint
SELECT
    CONSTRAINT_NAME,
    TABLE_NAME,
    COLUMN_NAME,
    REFERENCED_TABLE_NAME,
    REFERENCED_COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_NAME = 'Note' AND CONSTRAINT_NAME LIKE 'FK_%';
```

Part 7: Security Considerations

Security Features Implemented:

1. **Database-Level Isolation:**
   - Foreign key constraints ensure data integrity
   - Queries filter by UserId at the database level

2. **API-Level Authorization:**
   - Current user extracted from JWT token
   - User ownership validated before operations
   - Admin role bypasses user isolation
3. **Automatic User Assignment:**
   - Notes automatically assigned to current user
   - Client cannot manipulate user ownership
4. **Token-Based Security:**
   - User ID comes from cryptographically signed JWT
   - Claims cannot be forged by malicious clients

Security Best Practices:

```
// ✅ Good: Get user from token
var userId = currentUserHelper.GetCurrentUserId();

// ❌ Bad: Trust user ID from client
// var userId = noteViewModel.UserId; // Never do this!

// ✅ Good: Always assign to current user
noteViewModel.UserId = userId.Value;

// ✅ Good: Filter queries by user
query = query.Where(n => n.UserId == userId);
```

Part 8: Common Issues and Solutions

Issue 1: "Cannot add or update a child row: a foreign key constraint fails"

Cause: Trying to create a note with invalid UserId **Solution:** Always set `noteViewModel.UserId = currentUserId`

Issue 2: "User sees no notes after login"

Cause: No notes assigned to the user **Solution:** Create notes through the API (which auto-assigns user)

Issue 3: "Admin cannot see all notes"

Cause: Admin role not properly configured **Solution:** Verify JWT token contains correct role claim

Issue 4: "Migration fails with existing data"

Cause: Existing notes have no UserId **Solution:** Run data migration script to assign existing notes to users

---

Part 9: Database Migration Handling

Handling Existing Data

If you have existing notes without UserId values, create a data migration:

```sql
-- Option 1: Assign all existing notes to first user
UPDATE Note
SET UserId = (SELECT TOP 1 Id FROM Users ORDER BY Id)
WHERE UserId IS NULL;

-- Option 2: Create a "Legacy" user for orphaned notes
INSERT INTO Users (Email, Username, Password, Role, Firstname, Lastname)
VALUES ('legacy@system.com', 'legacy', 'hashed_password', 'User', 'Legacy',
'User');

UPDATE Note
SET UserId = (SELECT Id FROM Users WHERE Username = 'legacy')
WHERE UserId IS NULL;
```

Migration Script Template

```bash
# Create the migration
dotnet ef migrations add AddUserIdToNotes

# Review the generated migration file
# Add custom SQL if needed for existing data

# Apply the migration
dotnet ef database update
```

Conclusion

You have successfully implemented comprehensive user-based note isolation! The system now provides:

🔒 Security Benefits:

- Users can only access their own notes
- Admins have full access for management
- Database constraints prevent orphaned records
- JWT-based authentication ensures user identity

🏗 Architecture Benefits:

- Clean separation of user data
- Scalable multi-tenant design
- Proper foreign key relationships

- Role-based access control

📊 Performance Benefits:

- Database indexes on UserId for fast queries
- Efficient filtering at the database level
- Optimized search queries per user

🛡 Data Integrity:

- Foreign key constraints
- Automatic user assignment
- Cascade deletion handling
- Audit trail capabilities

Part 4: Update Data Seeding for Multi-User Testing

Step 16: Update DataSeeder for User-Specific Notes

File: `ListKeeper.ApiService/Data/DataSeeder.cs`

To properly test the user isolation functionality, update the DataSeeder to:

1. Create a John Doe user alongside the Admin user
2. Assign all sample notes to the John Doe user

Update the SeedUsersAsync Method:

Add the John Doe user creation after the Admin user:

```csharp
// Create the John Doe user database entity directly.
var johnDoeUser = new User
{
    Username = "john.doe",
    Email = "john.doe@email.com",
    Password = userHashedPassword,
    Role = "User",
    Firstname = "John",
    Lastname = "Doe"
};

// Add both users to the DbContext and save them.
await dbContext.Users.AddAsync(adminUser);
await dbContext.Users.AddAsync(johnDoeUser);
await dbContext.SaveChangesAsync();

logger.LogInformation("Admin user and John Doe user seeded successfully.");
```

Update the SeedNotesAsync Method:

Modify the notes seeding to assign all notes to John Doe:

```
// Find the John Doe user to assign notes to
var johnDoeUser = await dbContext.Users
    .FirstOrDefaultAsync(u => u.Username == "john.doe");

if (johnDoeUser == null)
{
    logger.LogError("John Doe user not found. Cannot seed notes without a user to assign them to.");
    return;
}

logger.LogInformation("Found John Doe user with ID: {UserId}. Assigning all sample notes to this user.", johnDoeUser.Id);
```

Then for each Note entity, add the UserId property:

```
new Note
{
    Title = "Finalize quarterly report",
    Content = "Compile sales data and performance metrics for the Q2 report. Draft slides for the presentation on Friday.",
    DueDate = new DateTime(2025, 7, 15, 17, 0, 0, DateTimeKind.Utc),
    IsCompleted = true,
    Color = "#D1E7DD",
    UserId = johnDoeUser.Id  // 👈 Add this to ALL notes
},
```

Testing the Multi-User Setup:

After updating and running the application:

1. **Login as John Doe:**
    - Username: `john.doe`
    - Password: `JohnPassword123!`
    - Should see all 20 sample notes
2. **Login as Admin:**
    - Username: `Admin`
    - Password: `AdminPassword123!`
    - Should see all notes from all users (currently John's notes)
3. **Create new notes as each user:**
    - John Doe will only see his notes
    - Admin will see notes from all users

This setup provides a perfect testing environment to demonstrate user isolation functionality! 🎯

Key Concepts Learned:

1. **Multi-Tenant Architecture:** Isolating user data in shared database
2. **Entity Relationships:** One-to-many relationships with foreign keys
3. **JWT Claims:** Extracting user context from authentication tokens
4. **Repository Pattern:** User-specific data access methods
5. **API Security:** Validating user ownership at endpoint level
6. **Database Migrations:** Schema changes with existing data
7. **Role-Based Access:** Different behavior for different user roles
8. **Data Seeding:** Creating proper test data for multi-user scenarios

This implementation provides a solid foundation for secure, scalable multi-user applications! 🎉