ListKeeper User Backend Documentation

Executive Summary

The ListKeeper.ApiService implements a comprehensive user management system using a layered architecture pattern with JWT-based authentication. The system follows Domain-Driven Design principles with clear separation of concerns across Presentation (Endpoints), Business Logic (Services), and Data Access (Repository) layers.

Architecture Overview

The user management system is built using the following architectural patterns:

- **Repository Pattern**: Abstracts data access logic
- **Service Layer Pattern**: Contains business logic and domain rules
- **Dependency Injection**: Manages service lifetimes and dependencies
- **Extension Methods**: Provides clean mapping between domain and view models
- **JWT Authentication**: Stateless token-based authentication
- **Auditing**: Automatic tracking of entity changes

Core Components

1. Models and Data Structures

User Domain Model (`Models/User.cs`)

The core domain entity representing a user in the system:

```
[Table("Users")]
public class User : IAuditable
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [Required]
    [EmailAddress]
    [StringLength(450)]
    public string Email { get; set; }

    [Required]
    [StringLength(450)]
    public string Password { get; set; } // Stores hashed password

    [StringLength(255)]
    public string? Role { get; set; }

    [StringLength(255)]
    public string? Username { get; set; }

    [StringLength(255)]
    public string? Firstname { get; set; }

    [StringLength(255)]
    public string? Lastname { get; set; }

    [StringLength(255)]
    public string? Phone { get; set; }

    // Audit fields from IAuditable
    public DateTime? CreatedAt { get; set; }
    public string? CreatedBy { get; set; }
    public DateTime? UpdatedAt { get; set; }
    public string? UpdatedBy { get; set; }
    public DateTime? DeletedAt { get; set; }
    public string? DeletedBy { get; set; }

    [NotMapped]
    public string Token { get; set; } // JWT token (not persisted)
}
```

Key Features:

- Implements `IAuditable` interface for automatic audit trail tracking
- Password field stores HMACSHA256 hashed passwords
- Token field is not mapped to database (transient)
- Email field has unique constraint capabilities
- Supports soft delete through `DeletedAt` field

UserViewModel (`Models/ViewModels/UserViewModel.cs`)

Data Transfer Object for API communication:

```
public class UserViewModel
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string Password { get; set; } // Input only, never returned with hash
    public string? Role { get; set; }
    public string? Username { get; set; }
    public string? Firstname { get; set; }
    public string? Lastname { get; set; }
    public string? Phone { get; set; }
    public DateTime? CreatedAt { get; set; }
    public string? CreatedBy { get; set; }
    public DateTime? UpdatedAt { get; set; }
    public string? UpdatedBy { get; set; }
    public DateTime? DeletedAt { get; set; }
    public string? DeletedBy { get; set; }
    public string Token { get; set; } // JWT token for authenticated responses
}
```

LoginViewModel (`Models/ViewModels/LoginViewModel.cs`)

Simplified model for authentication requests:

```
public class LoginViewModel
{
    [Required]
    public string Username { get; set; }

    [Required]
    public string Password { get; set; }
}
```

IAuditable Interface (`Models/Interfaces/IAuditable.cs`)

Contract for entities that support audit tracking:

```
public interface IAuditable
{
    DateTime? CreatedAt { get; set; }
    string? CreatedBy { get; set; }
    DateTime? UpdatedAt { get; set; }
    string? UpdatedBy { get; set; }
    DateTime? DeletedAt { get; set; }
    string? DeletedBy { get; set; }
}
```

2. Data Access Layer

DatabaseContext (`Data/DatabaseContext.cs`)

Entity Framework Core context with automatic auditing:

```
public class DatabaseContext : DbContext
{
    private readonly ILogger<DatabaseContext> _logger;
    private readonly IHttpContextAccessor _httpContextAccessor;

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

    // Automatic audit field population
    private void UpdateAuditableEntities()
    {
        var currentTime = DateTime.UtcNow;
        string userName =
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.Name)?.Value ??
"System";

        var modifiedEntities = ChangeTracker.Entries()
            .Where(e => e.Entity is IAuditable &&
                       (e.State == EntityState.Added || e.State ==
EntityState.Modified || e.State == EntityState.Deleted))
            .ToList();

        foreach (var entry in modifiedEntities)
        {
            var entity = (IAuditable)entry.Entity;

            if (entry.State == EntityState.Added)
            {
                entity.CreatedAt = currentTime;
                entity.CreatedBy = userName;
                entity.UpdatedAt = currentTime;
                entity.UpdatedBy = userName;
            }
            else if (entry.State == EntityState.Modified)
            {
                entity.UpdatedAt = currentTime;
                entity.UpdatedBy = userName;
            }
            else if (entry.State == EntityState.Deleted)
            {
                // Implement soft delete
                entry.State = EntityState.Modified;
                entity.DeletedAt = currentTime;
                entity.DeletedBy = userName;
            }
        }
    }
}
```

Key Features:

- Automatic audit field population on save operations
- Soft delete implementation for User entities
- Integration with HttpContext for user tracking
- Comprehensive error logging

IUserRepository Interface (`Data/IUserRepository.cs`)

Repository contract defining data access operations:

```
public interface IUserRepository
{
    Task<User> AddAsync(User user);
    Task<User?> AuthenticateAsync(string username, string password);
    Task<bool> Delete(User user);
    Task<bool> Delete(int id);
    Task<IEnumerable<User>> GetAllAsync();
    Task<User?> GetByIdAsync(int id);
    Task<User?> GetByUsernameAsync(string username);
    Task<User> Update(User user);
}
```

UserRepository Implementation (`Data/UserRepository.cs`)

Concrete implementation with JWT token generation:

```
public class UserRepository : IUserRepository
{
    private readonly DatabaseContext _context;
    private readonly ILogger<UserRepository> _logger;
    private readonly IConfiguration _configuration;

    // Authentication with JWT generation
    public async Task<User?> AuthenticateAsync(string username, string password)
    {
        var user = await _context.Users.SingleOrDefaultAsync(u => u.Username ==
username);

        if (user == null || password != user.Password)
        {
            return null; // Authentication failed
        }

        // Generate JWT token
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.ASCII.GetBytes(_configuration["Jwt:Secret"]!);

        var tokenDescriptor = new SecurityTokenDescriptor
        {
            Subject = new ClaimsIdentity(new[]
            {
                new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                new Claim(ClaimTypes.Name, user.Username!),
                new Claim(ClaimTypes.Role, user.Role ?? "User")
            }),
            Expires = DateTime.UtcNow.AddHours(1),
            Issuer = _configuration["Jwt:Issuer"],
            Audience = _configuration["Jwt:Audience"],
            SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
        };

        var token = tokenHandler.CreateToken(tokenDescriptor);
```

```
            user.Token = tokenHandler.WriteToken(token);

            return user;
        }
    }
```

Key Repository Features:

- Comprehensive CRUD operations
- JWT token generation during authentication
- Soft delete support with `DeletedAt` filtering
- Detailed error logging and exception handling
- Password comparison using hashed values

3. Business Logic Layer

IUserService Interface (`Services/IUserService.cs`)

Service contract defining business operations:

```
public interface IUserService
{
    Task<UserViewModel?> AuthenticateAsync(LoginViewModel loginViewModel);
    Task<UserViewModel?> CreateUserAsync(UserViewModel createUserVm);
    Task<bool> DeleteUserAsync(int id);
    Task<bool> DeleteUserAsync(UserViewModel userVm);
    Task<IEnumerable<UserViewModel>> GetAllUsersAsync();
    Task<UserViewModel?> GetUserByIdAsync(int id);
    Task<UserViewModel?> LoginAsync(string email, string password);
    Task<UserViewModel?> UpdateUserAsync(UserViewModel userVm);
}
```

UserService Implementation (`Services/UserService.cs`)

Business logic implementation with security features:

```
public class UserService : IUserService
{
    private readonly IUserRepository _repo;
    private readonly ILogger<UserService> _logger;
    private readonly IConfiguration _config;

    // Password hashing using HMACSHA256
    private string HashPassword(string password)
    {
        var secret = _config["ApiSettings:UserPasswordHash"];
        if (string.IsNullOrEmpty(secret))
        {
            throw new InvalidOperationException("Password hashing secret is not
configured.");
        }

        using var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(secret));
        var hash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
        return Convert.ToBase64String(hash);
    }

    public async Task<UserViewModel?> CreateUserAsync(UserViewModel createUserVm)
    {
        if (createUserVm == null) return null;

        // Hash password before storage
        string hashedPassword = HashPassword(createUserVm.Password);

        var user = new User
        {
            Email = createUserVm.Email,
            Username = createUserVm.Username,
            Firstname = createUserVm.Firstname,
            Lastname = createUserVm.Lastname,
            Role = createUserVm.Role,
            Phone = createUserVm.Phone,
            Password = hashedPassword
        };

        var createdUser = await _repo.AddAsync(user);
        return createdUser?.ToViewModel();
    }
}
```

Key Service Features:

- HMACSHA256 password hashing with configurable secret
- Domain model to view model mapping using extension methods
- Business rule enforcement (password hashing, validation)
- Comprehensive error handling and logging
- Separation of authentication and authorization concerns

4. Model Mapping Extensions

UserMappingExtensions (`Models/Extensions/UserMappingExtensions.cs`)

Clean mapping between domain and view models:

```
public static class UserExtensions
{
    // Domain to ViewModel mapping
    public static UserViewModel? ToViewModel(this User? user)
    {
        if (user == null) return null;

        return new UserViewModel
        {
            Id = user.Id,
            Username = user.Username ?? string.Empty,
            Email = user.Email ?? string.Empty,
            Role = user.Role ?? string.Empty,
            Firstname = user.Firstname,
            Lastname = user.Lastname,
            Token = user.Token,
            CreatedAt = user.CreatedAt,
            CreatedBy = user.CreatedBy,
            UpdatedAt = user.UpdatedAt,
            UpdatedBy = user.UpdatedBy,
            DeletedAt = user.DeletedAt,
            DeletedBy = user.DeletedBy
            // Note: Password is intentionally excluded for security
        };
    }

    // ViewModel to Domain mapping
    public static User? ToDomain(this UserViewModel? viewModel)
    {
        if (viewModel == null) return null;

        return new User
        {
            Id = viewModel.Id,
            Username = viewModel.Username,
            Email = viewModel.Email,
            Role = viewModel.Role,
            Firstname = viewModel.Firstname,
            Lastname = viewModel.Lastname
            // Password and Token are handled separately for security
        };
    }
}
```

5. Presentation Layer

UserEndpoints (`EndPoints/UserEndpoints.cs`)

RESTful API endpoints with proper security:

```
public static class UserEndpoints
{
    public static RouteGroupBuilder MapUserApiEndpoints(this RouteGroupBuilder
group)
    {
        // Admin-only endpoints
        group.MapGet("/", GetAllUsers)
             .RequireAuthorization("Admin");

        group.MapGet("/{userId}", GetUser)
             .RequireAuthorization("Admin");

        group.MapPost("/", CreateUser)
             .RequireAuthorization("Admin");

        group.MapPut("/{userId}", UpdateUser)
             .RequireAuthorization("Admin");

        group.MapDelete("/{userId}", DeleteUser)
             .RequireAuthorization("Admin");

        // Public authentication endpoint
        group.MapPost("/Authenticate", Authenticate)
             .AllowAnonymous();

        return group;
    }
}
```

Endpoint Security Model:

- All CRUD operations require "Admin" role authorization
- Authentication endpoint is publicly accessible
- JWT token generation occurs at authentication
- Proper dependency injection using `[FromServices]` attribute
- Comprehensive error handling with appropriate HTTP status codes

Complete Request Flow Analysis

Authentication Flow (POST /api/users/Authenticate)

1. **External Application Request**

   ```
   POST /api/users/Authenticate
   Content-Type: application/json

   {
     "username": "admin@example.com",
     "password": "AppleRocks!"
   }
   ```

2. **UserEndpoints.Authenticate Method**
   - Receives `LoginViewModel` from request body
   - Dependency injection provides `IUserService`, `ILoggerFactory`, and `IConfiguration`

- Calls `userService.AuthenticateAsync(model)`

3. **UserService.AuthenticateAsync Method**

    - Validates input `LoginViewModel`
    - Calls internal `LoginAsync(loginViewModel.Username, loginViewModel.Password)`
    - Hashes the provided password using `HashPassword()` method
    - Calls `_repo.AuthenticateAsync(email, hashedPassword)`

4. **UserRepository.AuthenticateAsync Method**

    - Queries database: `_context.Users.SingleOrDefaultAsync(u => u.Username == username)`
    - Compares hashed password with stored hash
    - If valid, generates JWT token with user claims
    - Returns `User` domain object with populated `Token` property

5. **Back to UserService**

    - Receives `User` from repository
    - Calls `user?.ToViewModel()` extension method to convert to `UserViewModel`

6. **Back to UserEndpoints**

    - Receives `UserViewModel` from service
    - Generates new JWT token using `GenerateJwtToken()` method (overwrites repository token)
    - Returns `Results.Ok(user)` with populated token

7. **Extension Method Flow (ToViewModel)**

```
public static UserViewModel? ToViewModel(this User? user)
{
    return new UserViewModel
    {
        Id = user.Id,
        Username = user.Username ?? string.Empty,
        Email = user.Email ?? string.Empty,
        Role = user.Role ?? string.Empty,
        Token = user.Token,
        // Password field is intentionally excluded
        // Other audit fields mapped
    };
}
```

8. **Response to External Application**

```
{
  "id": 1,
  "username": "Admin",
  "email": "admin@example.com",
  "role": "Admin",
  "firstname": "Admin",
  "lastname": "User",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "createdAt": "2025-07-03T10:00:00Z",
  "createdBy": "System"
}
```

User Creation Flow (POST /api/users)

1. **External Application Request** (Requires JWT token with Admin role)

```
POST /api/users
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json

{
  "username": "newuser",
  "email": "newuser@example.com",
  "password": "SecurePassword123",
  "role": "User",
  "firstname": "John",
  "lastname": "Doe"
}
```

2. **Authentication/Authorization Middleware**
   - Validates JWT token
   - Extracts user claims
   - Verifies "Admin" role requirement
3. **UserEndpoints.CreateUser Method**
   - Receives `UserViewModel` from request body
   - Calls `userService.CreateUserAsync(model)`
4. **UserService.CreateUserAsync Method**
   - Validates input `UserViewModel`
   - Hashes password using `HashPassword(createUserVm.Password)`
   - Creates new `User` domain object with hashed password
   - Calls `_repo.AddAsync(user)`
5. **UserRepository.AddAsync Method**
   - Adds user to `_context.Users`
   - Calls `_context.SaveChangesAsync()`
6. **DatabaseContext.SaveChangesAsync**

- Triggers `UpdateAuditableEntities()` method
- Populates audit fields (CreatedAt, CreatedBy, UpdatedAt, UpdatedBy)
- Extracts current user from HttpContext claims
- Saves to database with auto-generated ID

7. **Back to UserService**

   - Receives created `User` with populated ID
   - Calls `createdUser?.ToViewModel()` extension method

8. **Back to UserEndpoints**

   - Returns `Results.Created($"/api/users/{newUser.Id}", newUser)`

9. **Extension Method Flow (ToViewModel)**

   - Maps all safe fields from domain to view model
   - Excludes password hash for security
   - Includes audit information

10. **Response to External Application**

```json
{
  "id": 2,
  "username": "newuser",
  "email": "newuser@example.com",
  "role": "User",
  "firstname": "John",
  "lastname": "Doe",
  "token": "",
  "createdAt": "2025-07-03T14:30:00Z",
  "createdBy": "Admin",
  "updatedAt": "2025-07-03T14:30:00Z",
  "updatedBy": "Admin"
}
```

User Retrieval Flow (GET /api/users/{userId})

1. **External Application Request**

```http
GET /api/users/2
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

2. **UserEndpoints.GetUser Method**

   - Extracts `userId` from route parameter
   - Calls `userService.GetUserByIdAsync(userId)`

3. **UserService.GetUserByIdAsync Method**

   - Calls `_repo.GetByIdAsync(id)`

4. **UserRepository.GetByIdAsync Method**

   - Executes `_context.Users.FindAsync(id)`
   - Returns `User` domain object or null

5. **Back to UserService**

   - Calls `user?.ToViewModel()` extension method

6. **Back to UserEndpoints**

   - Returns `Results.Ok(user)` or `Results.NotFound()`

7. **ToDomain Extension Method Flow** (for Update operations)

```
public static User? ToDomain(this UserViewModel? viewModel)
{
    return new User
    {
        Id = viewModel.Id,
        Username = viewModel.Username,
        Email = viewModel.Email,
        Role = viewModel.Role,
        Firstname = viewModel.Firstname,
        Lastname = viewModel.Lastname
        // Password and Token excluded - handled separately
    };
}
```

Configuration and Security

Application Configuration (`appsettings.json`)

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\MSSQLLocalDB;;Database=ListKeeperData;Trusted_Connection=True;MultipleA
ctiveResultSets=true;TrustServerCertificate=True"
  },
  "Jwt": {
    "Issuer": "https://api.listkeeper.com",
    "Audience": "https://listkeeper.com",
    "Secret": "w8z/C?F)J@NcRfUjXn2r5u7x!A%D*G-KaPdSgVkYp3s6v9y$B&E)H+MbQeThWmZq"
  },
  "ApiSettings": {
    "RoutePrefix": "/api",
    "ServiceName": "ApiService",
    "UserPasswordHash": "G-KaPdSgVkYp3s6v9y$B&E)H+MbQeThWmZq"
    }
}
```

Security Implementation

1. **Password Security**
    - HMACSHA256 hashing with configurable secret key
    - Never store plain text passwords
    - Separate hashing secret from JWT secret
2. **JWT Token Security**
    - 256-bit secret key requirement
    - 1-hour token expiration
    - Role-based claims for authorization
    - Symmetric key signing
3. **Authorization Policies**
    - "Admin" policy requires Admin role in JWT claims
    - All user management operations require Admin authorization
    - Authentication endpoint publicly accessible
4. **Data Security**
    - Password hashes never returned in API responses
    - Audit trail for all user operations
    - Soft delete implementation

Data Seeding (`Data/DataSeeder.cs`)

The system includes automatic seeding of an admin user:

```
public static async Task SeedAdminUserAsync(IHost app)
{
    // Creates admin user if no users exist
    var adminUser = new User
    {
        Username = "Admin",
        Email = "admin@example.com",
        Password = hashedPassword, // "AppleRocks!" hashed
        Role = "Admin",
        Firstname = "Admin",
        Lastname = "User"
    };
}
```

Dependency Injection Configuration (`Program.cs`)

```
// Database Context
builder.Services.AddDbContext<DatabaseContext>(options =>
    options.UseSqlServer(connectionString));

// Repository and Service Registration
builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IUserService, UserService>();

// JWT Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey)),
            ValidateLifetime = true,
            ClockSkew = TimeSpan.FromMinutes(1)
        };
    });

// Authorization Policies
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Admin", policy => policy.RequireRole("Admin"));
});
```

API Endpoints Summary

Endpoint	Method	Auth Required	Description
`/api/users`	GET	Admin	Get all users
`/api/users/{id}`	GET	Admin	Get specific user
`/api/users`	POST	Admin	Create new user
`/api/users/{id}`	PUT	Admin	Update existing user
`/api/users/{id}`	DELETE	Admin	Delete user (soft delete)
`/api/users/Authenticate`	POST	None	Authenticate user and get token

Error Handling Strategy

1. **Service Layer**: Business logic validation and logging
2. **Repository Layer**: Data access error handling and logging
3. **Endpoint Layer**: HTTP status code mapping and user-friendly messages
4. **Global**: Exception middleware for unhandled exceptions

Performance Considerations

1. **Database Queries**: Entity Framework Core with async operations
2. **Password Hashing**: HMACSHA256 for reasonable performance vs. security balance
3. **JWT Tokens**: Stateless authentication reduces server memory usage
4. **Dependency Injection**: Scoped lifetime for web request optimization

Recommended Improvements

1. **Security Enhancements**:
    - Replace HMACSHA256 with BCrypt/Argon2 for password hashing
    - Implement password complexity requirements
    - Add rate limiting for authentication attempts
    - Enable JWT audience/issuer validation
2. **Performance Optimizations**:
    - Add caching for frequently accessed users
    - Implement pagination for user lists
    - Add database indexing on username/email
3. **Feature Additions**:
    - Email verification for new users
    - Password reset functionality
    - User profile management endpoints
    - Role management system

This comprehensive user management system provides a solid foundation for authentication and authorization in the ListKeeper application, with proper separation of concerns, security best practices, and extensible architecture.