ITERATION 7

OUR CLIENT:

EPI-USE

groupelephant.com
BEYOND CORPORATE PURPOSE

TEAM 35

CANVAS

NGO PROJECT PORTFOLIO MANAGEMENT SYSTEM

This document defines the technical narratives of all the use cases identified for the proposed Integrated NGO Management System for EPI-USE and groupelephant.com.



Amahle Khoza

u22539702@tuks.co.za

+27 71 023 1872



Stephan Bredenhann

u23667436@tuks.co.za

+27 71 566 0997



Christopher Bailey

u24661092@tuks.co.za

+27 71 365 9963



Nicholas Bailey

u24671691@tuks.co.za

+27 71 610 6952



Alexander Steven

u22496786@tuks.co.za

+27 72 370 0225

Table of Contents

Table of Tables 7
1. Document Introduction 11
2. Technical Use Case Narratives 12
2.1 User Subsystem Use Case Narratives 12
2.1.1 User Login 12
2.1.2 User Logout 17
2.1.3 Reset Password 21
2.1.4 Create User 26
2.1.5 Update User 35
2.1.6 View User 41
2.1.7 Reactivate User 45
2.1.8 Deactivate User 49
2.1.9 View FAQ 53
2.2 Admin Subsystem Use Case Narratives 56
2.2.1 Create User Profile 56
2.2.2 Update User Profile 61
2.2.3 View User Profile 66
2.2.4 Deactivate User Profile 70
2.2.5 Add FAQ 75
2.2.6 Update FAQ 80
2.2.7 Remove FAQ 84
2.2.8 Add FAQ Type 88
2.2.9 Update FAQ Type 93
2.2.10 Remove FAQ Type 97
2.2.11 Create Role 101
2.2.12 View Role 106
2.2.13 Update Role 109
2.2.14 Reactivate Role 113
2.2.15 Deactivate Role 116
2.2.16 Update Role Permissions 120
2.2.17 View Role Permission 124
2.3 Inventory Subsystem Use Case Narratives 127
2.3.1 Create Procurement 127
2.3.2 View Procurement 136
2.3.3 View Inventory 141
2.3.4 Write-Off Inventory 149

2.3.5 Allocate Inventory 155
2.3.6 Create Inventory Item Type 161
2.3.7 View Inventory Item Types 166
2.3.8 Update Inventory Item Type 170
2.3.9 Deactivate Inventory Item Type 176
2.3.10 Reactivate Inventory Item Type 181
2.4 Group Subsystem Use Case Narratives 186
2.4.1 Create Group 186
2.4.2 View Groups 192
2.4.3 Update Group 197
2.4.4 Deactivate Group 204
2.4.5 Create Skill 208
2.4.6 View Skills 214
2.4.7 View Skill 217
2.4.8 Update Skill 220
2.4.9 Deactivate Skill 225
2.5 Warehouse Subsystem Use Case Narratives 229
2.5.1 Create Warehouse 229
2.5.2 View Warehouse 233
2.5.3 Update Warehouse 235
2.5.4 Reactivate Warehouse 238
2.5.5 Deactivate Warehouse 240
2.5.6 Inventory Stock Take 242
2.6 Vendor Subsystem Use Case Narratives 248
2.6.1 Create Vendor 248
2.6.2 View Vendor 253
2.6.3 View Vendors 257
2.6.4 Edit Vendor 261
2.6.5 Reactivate Vendor 266
2.6.6 Deactivate Vendor 270
2.6.7 Create Vendor Type 274
2.6.8 Update Vendor Type 279
2.6.9 View Vendor Types 283
2.6.10 Reactivate Vendor Type 287
2.6.11 Deactivate Vendor Type 291
2.7 Project Management Subsystem Use Case Narratives 295
2.7.1 Create Project 295
2.7.2 View Project 301

2.7.3 Update Project 304
2.7.4 Deactivate Project 307
2.7.5 Create Project Type 310
2.7.6 View Project Type 313
2.7.7 Update Project Type 315
2.7.8 Deactivate Project Type 318
2.7.9 Add Indemnity Form 321
2.7.10 Update Indemnity Form 325
2.7.11 View Indemnity Form 328
2.7.12 Remove Indemnity Form 330
2.7.13 Capture Indemnity Form - 332
2.7.14 Capture Project Feedback - 335
2.7.15 View Project Feedback - 338
2.7.16 Create Impact Feedback Form 341
2.7.17 View Impact Feedback Forms 344
2.7.18 Update Impact Feedback Forms 347
2.7.19 Deactivate Impact Feedback Forms 350
2.7.20 Capture Impact Feedback 353
2.7.21 View Impact Feedback 356
2.7.22 Add Photo 359
2.7.23 View Photo 362
2.7.24 Delete Photo 365
2.7.25 Add Photo Album 368
2.7.26 View Photo Album 371
2.7.27 Update Photo Album 373
2.7.28 Remove Photo Album 375

2.8 K9 Subsystem Use Case Narratives 378

2.8.1 Add Dog 378
2.8.2 View Dog 386
2.8.3 View Dogs 392
2.8.4 Update Dog 396
2.8.5 Deactivate Dog 404
2.8.6 Assign User to Dog 408
2.8.7 View User-Dog Assignment 414
2.8.8 Update User-Dog Assignment 418
2.8.9 Reactivate User-Dog Assignment 424
2.8.10 Deactivate User-Dog Assignment 429
2.8.11 Add K9 Role 433
2.8.12 Update K9 Role 438
2.8.13 View K9 Role 443

2.8.14 Reactivate K9 Role 447
2.8.15 Deactivate K9 Role 451
2.8.16 View K9 Ancestry 455
2.9 Rhino Subsystem Use Case Narratives 459
2.9.1 Add Rhino 459
2.9.2 View Rhino 466
2.9.3 View Rhinos 470
2.9.4 Update Rhino 476
2.9.5 Deactivate Rhino 483
2.10 Elephant Subsystem Use Case Narratives 487
2.10.1 Add Elephant 487
2.10.2 View Elephant 491
2.10.3 View Elephants 494
2.10.4 Update Elephant 496
2.10.5 Deactivate Elephant 499
2.11 Messages Subsystem Use Case Narratives 502
2.11.1 Create Message 502
2.11.2 View Messages 505
2.12 Report Subsystem Use Case Narratives 510
2.12.1 Generate Custom Report 510
2.12.2 View Custom Report 514
2.12.3 Update Custom Report 517
2.12.4 Remove Custom Report 520
2.12.5 View Bike Impact Report 523
2.12.6 View Budget Report 526
2.12.7 View Rhino Report 529
2.12.8 View Elephant Report 533
2.12.9 View K9 Report 537
2.12.10 View Machine Learning Dashboard 548
2.13 System Subsystem Use Case Narratives 554
2.13.1 Backup System 554
2.13.2 Restore System 560
2.13.3 View Audit Log 566
2.14 Budget Subsystem Use Case Narratives 571
2.14.1 View Project Budget 571
2.14.2 View Project Budgets 574
2.14.3 Log An Expense 577

2.14.4 Update An Expense 580
2.14.5 View Expense 583
2.14.6 Remove Expense 585
3. Conclusion 588
4. Sign-off by Client 589
5. Sign-off by Team 590

Table of Tables

Table 1: User Login - Technical Use Case Narrative....16
Table 2: User Logout - Technical Use Case Narrative....20
Table 3: Reset Password - Technical Use Case Narrative....25
Table 4: Create User - Technical Use Case Narrative....34
Table 5: Update User - Technical Use Case Narrative....40
Table 6: View User - Technical Use Case Narrative....44
Table 7:Reactivate User - Technical Use Case Narrative....48
Table 8: Deactivate User - Technical Use Case Narrative....52
Table 9: View FAQ - Technical Use Case Narrative....55
Table 10: Create User Profile - Technical Use Case Narrative....60
Table 11: Update User Profile - Technical Use Case Narrative....65
Table 12: View User Profile - Technical Use Case Narrative....69
Table 13: Deactivate User Profile – Technical Use Case Narrative....74
Table 14: Add FAQ - Technical Use Case Narrative....79
Table 15: Update FAQ - Technical Use Case Narrative....83
Table 16: Remove FAQ - Technical Use Case Narrative....87
Table 17: Add FAQ Type - Technical Use Case Narrative....92
Table 18: Update FAQ Type - Technical Use Case Narrative....96
Table 19: Remove FAQ Type - Technical Use Case Narrative....100
Table 20: Create Role - Technical Use Case Narrative....105
Table 21: View Role - Technical Use Case Narrative....108
Table 22: Update Role - Technical Use Case Narrative....112
Table 23: Reactivate Role - Technical Use Case Narrative....115
Table 24: Deactivate Role - Technical Use Case Narrative....119
Table 25: Update Role Permission - Technical Use Case Narrative....123
Table 26: View Role Permission - Technical Use Case Narrative....126
Table 27: Create Procurement - Technical Use Case Narrativ....135
Table 28:Table 27: View Procurement - Technical Use Case Narrative....140
Table 29: View Inventory - Technical Use Case Narrative....149
Table 30: Write-Off Inventory - Technical Use Case Narrative....154
Table 31: Allocate Inventory - Technical Use Case Narrative....160
Table 32: Create Inventory Item Type - Technical Use Case Narrative....165
Table 33: View Inventory Item Types - Technical Use Case Narrative....169
Table 35: Update Inventory Item Type - Technical Use Case Narrative....175
Table 36: Deactivate Inventory Item Type - Technical Use Case Narrative....180
Table 37: Receive Inventory Item Type - Technical Use Case Narrative....184
Table 38: Create Group - Technical Use Case Narrative....191
Table 39: View Groups - Technical Use Case Narrative....196
Table 40: Update Group - Technical Use Case Narrative....203

Table 41: Deactivate Group - Technical Use Case Narrative 207
Table 42: Create Skill - Technical Use Case Narrative.....213
Table 43: View Skills - Technical Use Case Narrative.....216
Table 44: View Skill - Technical Use Case Narrative 219
Table 45: Update Skill - Technical Use Case Narrative 224
Table 46: Deactivate Skill - Technical Use Case Narrative 228
Table 47: Create Warehouse - Technical Use Case Narrative 232
Table 48: View Warehouse - Technical Use Case Narrative 234
Table 49: Update Warehouse - Technical Use Case Narrative 237
Table 50: Reactivate Warehouse -Technical Use Case Narrative.....239
Table 51: Deactivate Warehouse - Technical Use Case Narrative.....241
Table 52: Inventory Stock Take - Technical Use Case Narrative.....247
Table 53: Create Vendor - Technical Use Case Narrative 252
Table 54: View Vendor - Technical Use Case Narrative.....256
Table 55: View Vendors - Technical Use Case Narrative 260
Table 56: Edit Vendor - Technical Use Case Narrative.....265
Table 57: Reactivate Vendor - Technical Use Case Narrative.....269
Table 58: Deactivate Vendor - Technical Use Case Narrative 273
Table 59: Create Vendor Type - Technical Use Case Narrative 278
Table 60: Update Vendor Types - Technical Use Case Narrative.....282
Table 61: View Vendor Types - Technical Use Case Narrative 286
Table 62: Reactivate Vendor Type - Technical Use Case Narrative 290
Table 63 :Deactivate Vendor Type - Technical Use Case Narrative.....294
Table 64: Create Project - Technical Use Case Narrative.....300
Table 65: View Project - Technical Use Case Narrative 303
Table 66: Update Project - Technical Use Case Narrative 306
Table 67: Deactivate Project - Technical Use Case Narrative 309
Table 68: Create Project Type - Technical Use Case Narrative.....312
Table 69: View Project Type - Technical Use Case Narrative 314
Table 70: Update Project Type - Technical Use Case Narrative 317
Table 71: Deactivate Project Type - Technical Use Case Narrative 320
Table 72: Add Indemnity Form - Technical Use Case Narrative 324
Table 73: Update Indemnity Form - Technical Use Case Narrative.....327
Table 74: View Indemnity Form - Technical Use Case Narrative.....329
Table 75: Remove Indemnity Form - Technical Use Case Narrative.....331
Table 76: Remove Indemnity Form - Technical Use Case Narrative.....334
Table 77: Capture Project Feedback - Technical Use Case Narrative 337
Table 78: View Project Feedback - Technical Use Case Narrative.....340
Table 79: Create Impact Feedback Form - Technical Use Case Narrative 343
Table 80: View Impact Feedback Forms - Technical Use Case Narrative 346
Table 81: Update Impact Feedback Form - Technical Use Case Narrative.....349

Table 82: Deactivate Impact Feedback Form - Technical Use Case Narrative 352
Table 83: Capture Impact Feedback- Technical Use Case Narrative 355
Table 84: View Impact Feedback - Technical Use Case Narrative 358
Table 85: Add Photo - Technical Use Case Narrative.....361
Table 86: View Photo - Technical Use Case Narrative 364
Table 87: Delete Photo - Technical Use Case Narrative 367
Table 88: Add Photo Album - Technical Use Case Narrative 370
Table 89: View Photo Album - Technical Use Case Narrative.....372
Table 90: Update Photo Album - Technical Use Case Narrative.....374
Table 91: Remove Photo Album - Technical Use Case Narrative 376
Table 92: Add Dog - Technical Use Case Narrative.....385
Table 93: View Dog - Technical Use Case Narrative 391
Table 94: View Dogs - Technical Use Case Narrative.....395
Table 95: Update Dog - Technical Use Case Narrative 403
Table 96: Deactivate Dog - Technical Use Case Narrative 407
Table 97: Assign User to Dog - Technical Use Case Narrative 413
Table 98: View User-Dog Assignment - Technical Use Case Narrative.....417
Table 99: Update User-Dog Assignment - Technical Use Case Narrative.....423
Table 100: Reactivate User-Dog Assignment - Technical Use Case Narrative.....428
Table 101: Deactivate User-Dog Assignment - Technical Use Case Narrative 432
Table 102: Add K9 Role - Technical Use Case Narrative 437
Table 103: Update K9 Role - Technical Use Case Narrative.....442
Table 104: View K9 Roles - Technical Use Case Narrative 446
Table 105: Reactivate K9 Role - Technical Use Case Narrative 450
Table 106: Deactivate K9 Role - Technical Use Case Narrative 454
Table 107: View K9 Ancestry - Technical Use Case Narrative.....458
Table 108: Add Rhino - Technical Use Case Narrative.....465
Table 109: View Rhino - Technical Use Case Narrative 469
Table 110: View Rhino - Technical Use Case Narrative 475
Table 111: Update Rhino - Technical Use Case Narrative 482
Table 112: 9.5 Deactivate Rhino - Technical Use Case Narrative 486
Table 113: Add Elephant - Technical Use Case Narrative.....490
Table 114: View Elephant - Technical Use Case Narrative 492
Table 115: View Elephants - Technical Use Case Narrative.....495
Table 116: Update Elephant - Technical Use Case Narrative 498
Table 117: Deactivate Elephant - Technical Use Case Narrative 501
Table 118: Create Message - Technical Use Case Narrative.....504
Table 119: View Messages - Technical Use Case Narrative.....509
Table 120: Generate Custom Report - Technical Use Case Narrative 513
Table 121: View Custom Report - Technical Use Case Narrative 516
Table 122: Update Custom Report - Technical Use Case Narrative 519

Table 123: Remove Custom Report - Technical Use Case Narrative 522
Table 124: View Bike Impact Report - Technical Use Case Narrative.....525
Table 125: View Budget Report - Technical Use Case Narrative 528
Table 126: View Rhino Report - Technical Use Case Narrative 532
Table 127: View Elephant Report - Technical Use Case Narrative 536
Table 128: View K9 Report - Technical Use Case Narrative.....547
Table 129: View Machine Learning Dashboard - Technical Use Case Narrative 553
Table 130: Backup System - Technical Use Case Narrative 559
Table 131: Restore System - Technical Use Case Narrative 565
Table 132: View Audit Log - Technical Use Case Narrative 570
Table 133: View Project Budget - Technical Use Case Narrative.....573
Table 134: View Project Budgets - Technical Use Case Narrative 576
Table 135: Log An Expense – Technical Use Case Narrative.....579
Table 136: Update An Expense - Technical Use Case Narrative 582
Table 137: View Expense - Technical Use Case Narrative 584
Table 138: Remove Expense - Technical Use Case Narrative 587

1. Document Introduction

This document outlines the technical narratives of all the use cases defined for the proposed Integrated NGO Management System for EPI-USE and groupelephant.com.

2. Technical Use Case Narratives

2.1 User Subsystem Use Case Narratives

2.1.1 User Login

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Stephan Brednehnn	**Date:** 28 April 2025 **Version:** 1.0	
USE CASE NAME:	User Login	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	1.1		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	User		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	Administrator		
Description:	This use case describes the event when an existing user logs onto the system to view and manage their account and projects. The user enters their credentials into the Angular frontend. The system evaluates the reactive form, and upon submission, routes the JSON payload to the corresponding ".NET 8" API endpoint ("POST api/Auth/login"). The backend executes a SQL Read Query via Entity Framework Core on the "User" Entity to validate the entered credentials and account status using the "PasswordHasherService". The system generates a JWT token embedded with relevant claims and calculates the user's onboarding progression. The use case concludes with the frontend persisting the JWT token in Local Storage, redirecting the user to the appropriate page, and displaying a success toast notification.		
Pre-condition:	• The user must be registered on the system, and their account must be active.		
Trigger:	The user requests to log into their account.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

INTEGRATED NGO MANAGEMENT SYSTEM			
	Step 1: The user requests to log into their account by navigating to the base URL or "/login" route.		**Step 2:** The Angular Router loads the Login Component. The system renders the user interface containing the following elements on a centralized authentication card: • A Login header reading "Sign In". • An Angular Reactive Form Group containing an Email label and an input field with email formatting validation. • A Password label and an obfuscated input field. • A hyperlink labelled "Forgot password?". • A primary "Sign In" submit button • A label reading "Don't have an Account?". • A hyperlink labelled "Sign Up Here".
	Step 3: The user captures their email address and password into the respective input fields. And selects the "sign In" button. [ALT]		**Step 4**: The system captures the form data and the Angular AuthService sends a POST request to the api/Auth/login endpoint with a LoginRequest JSON payload containing the email and password.

INTEGRATED NGO MANAGEMENT SYSTEM			
			The API's AuthController receives the request. The system connects to the ApplicationDbContext and executes an asynchronous query (SingleOrDefaultAsync) against the following entity to locate a matching record by UserEmail: **User Entity:** • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserEmail • PasswordHash • UserStatus • UserAllergies • ProfilePicture • ActiveStatus • UserShirtSize • SkillsOnboardingCompleted
			Step 5: The system validates the retrieved entity against the provided credentials and business rules before proceeding. The system performs the following checks: • Verifies that the user record was found.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Invokes the PasswordHasherService.VerifyPassword() method, hashing the incoming password string and validating it against the PasswordHash field stored in the database. • Verifies that ActiveStatus is true (the account is not deactivated). [ALT]
			Step 6: Upon successful validation, the system delegates to the "JwtTokenService" to generate a secure JWT token. The token embeds standard claims alongside the user's ID and Email, granting the user access to authorized endpoints.
			Step 6: The system confirms that the user has completedOnboarding in the User table. [ALT]
			Step 7: The JWT is stored in the user's browser local storage, and they are redirected to the Primary System Dashboard.
Alternate courses:	**[ALT] Step 3:**		

INTEGRATED NGO MANAGEMENT SYSTEM	
	The user forgets their login details and selects the "Forgot Password" hyperlink, which triggers the Angular Router to navigate to the password reset component, invoking use case 1.3 Reset Password.
	[ALT] Step 5: The system fails to locate the user's email in the database, or the "PasswordHasherService" returns false for the password check. The API immediately aborts the flow and returns an "HTTP 401 Unauthorized" response with the message "Invalid credentials." The frontend catches the error and renders a red error toast message reading "Invalid credentials, please try again."
	[ALT] Step 7: The system identifies that the user's "ActiveStatus" is false. The API halts the login process and returns an "HTTP 403 Forbidden" response along with a JSON payload indicating "accountDeactivated = true". The frontend intercepts this specific error and invokes Use Case 1.7 Reactivate User.
Conclusion:	The use case concludes when the system successfully validates the credentials, issues a JWT token and routes the user to the correct application view.
Post-condition:	The user is actively authenticated within the system, holds a valid JWT token in local storage, and is permitted access to secured application features.
Business Rules:	• Secure Access
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.
Assumptions:	• None
Open issues:	• None

Table 1: User Login - Technical Use Case Narrative

2.1.2 User Logout

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Stephan Brednehnn	**Date:** 28 April 2025 **Version:** 1.0	
USE CASE NAME:	User Logout	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	1.2		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	User		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	Administrator		
Description:	This use case occurs when an authenticated user deliberately chooses to terminate their active session. The frontend initiates the process by calling the secured “POST api/Auth/logout” API endpoint. The backend intercepts the user's Bearer token, extracts the unique JWT identifier (“jti”) and expiration time (“exp”), and registers the token with the “TokenRevocationService” to forcibly invalidate it before its natural expiration. Finally, the frontend purges local state and navigates the user securely back to the login interface.		
Pre-condition:	• The user must be currently authenticated with a valid session and actively navigating the system dashboard or internal pages.		
Trigger:	The user clicks the "Logout" button located in the application's global navigation bar or user profile dropdown.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user opens the profile dropdown menu in the top navigation bar and clicks the "Logout" button.		**Step 2:** The Angular frontend intercepts the click event. The “AuthService” extracts the current JWT token from Local Storage and dispatches a secured

INTEGRATED NGO MANAGEMENT SYSTEM			
			"POST" request to the "api/Auth/logout" endpoint, passing the token in the "Authorization: Bearer" header.
			Step 3: The API's "AuthController" receives the request. Because the endpoint is decorated with the "\[Authorize]" attribute, the ASP.NET Core middleware first validates that the token itself is structurally valid and not expired.
			Step 4: The controller manually parses the "Authorization" header using "JwtSecurityTokenHandler.ReadJwtToken()". The system queries the token's embedded claims to extract the "jti" (JWT ID) and the "exp" (Expiration Timestamp) values.
			Step 5: The system invokes the "TokenRevocationService.Revoke()" method, passing in the extracted "jti" and a computed "expiresAt" "DateTimeOffset". The service adds this unique token ID to a

INTEGRATED NGO MANAGEMENT SYSTEM			
			secure, server-side blocklist (revocation list) to guarantee the token can no longer be used for future authenticated requests, effectively neutralizing it immediately.
			Step 6: The API returns an "HTTP 200 OK" JSON response with a success message reading "Logged out."
			Step 7: The Angular "AuthService" processes the successful HTTP response. The system executes a localized cleanup routine, permanently deleting the JWT token from the browser's Local Storage and clearing any cached user profile information.
			Step 8: The Angular Router executes a programmatic navigation, redirecting the user entirely out of the secure zone and rendering the Login Component interface.
Alternate courses:	None		
Conclusion:	The active session is securely terminated on both the client-side browser and the backend server.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Post-condition:	The user's access to secured endpoints is completely revoked until a new, successful authentication cycle is completed.
Business Rules:	• Secure Access
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 2: User Logout - Technical Use Case Narrative

2.1.3 Reset Password

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Stephan Brednehnn</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Reset Password</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.3</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">Administrator</td></tr>
<tr><td>Description:</td><td colspan="3">This use case handles account recovery when a user has forgotten their password or requires a security reset. The user requests a reset code via the frontend, which triggers the backend to generate a cryptographically secure 6-digit PIN. This PIN is hashed, stored in the database (“PasswordResetCode” Entity), and dispatched to the user via the “EmailSenderService”. The user then enters the PIN alongside a new password into a validation form. The system verifies the code, enforces retry limits, updates the “PasswordHash” on the “User” Entity, and finalizes the recovery process.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be attempting to log in, and their email address must be registered on the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "Forgot Password?" hyperlink located on the login page.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "Forgot Password?" link.</td><td></td><td>Step 2: The Angular Router unmounts the Login Component and mounts the Password Reset Request Component. The UI</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			renders a dedicated card displaying a sub-header, a descriptive instruction text, a Reactive Form with an Email input field, a "Back to Login" link, and a "Send Reset Code" primary button.
	Step 3: The user enters their email address and clicks the "Send Reset Code" button.		**Step 4:** The frontend's Reactive Form validates the email format. The system dispatches a “POST” request to “api/Auth/forgot password/request” containing the “ForgotPasswordRequestDto” JSON payload.
			Step 5: The backend receives the request. The system connects to the database via Entity Framework and performs a query against the “User” table to determine if an account matching the provided email address exists. [ALT]
			Step 6: The system generates a random, secure 6-digit numeric string using the “RandomNumberGenerator” function.
			Step 7: The system calculates a secure

INTEGRATED NGO MANAGEMENT SYSTEM			
			hash of the generated code. It creates a 16-byte cryptographic salt, concatenates the code and the salt, and hashes the resulting string using the "SHA256" algorithm.
			Step 8: The system initializes a new "PasswordResetCode" Entity, populating it with the user's ID, the generated "CodeSalt", the "CodeHash", a "CreatedAt" timestamp, an "ExpiresAt" timestamp strictly set to 10 minutes in the future, and initializes the "FailedAttempts" integer to 0.
			Step 9: The system invokes the "EmailSenderService.SendPasswordResetCodeAsync()" method, commanding the external Email Provider (Resend)to dispatch an automated email containing the raw 6-digit code to the user's email address.
			Step 10: The API returns an "HTTP 200 OK" response with a standard, non-

INTEGRATED NGO MANAGEMENT SYSTEM			
			revealing message: "If the email exists, a code has been sent."
	Step 11: The user retrieves the 6-digit code from their email client, enters it into the code input field.		**Step 12:** The system validates the provided code. It concatenates the raw string provided by the user with the "CodeSalt" retrieved from the database, runs the "SHA256" hashing algorithm, and performs a comparison against the stored "CodeHash". The Api returns A 'HTTP 200 OK' response. [ALT]
			Step 13: The Angular frontend receives the response and dynamically transitions the UI state. It hides the code input field and displays the new password and confirm password fields.
	Step 14: The user enters their desired new password and re-enters it in the required fields to ensure it is accurately captured.		**Step 15:** The system overwrites the "PasswordHash" property on the "User" Entity with the newly generated hash. Concurrently, the system updates the "PasswordResetCode" Entity, populating the "UsedAt" property with the current UTC timestamp to

INTEGRATED NGO MANAGEMENT SYSTEM			
			permanently expire the code.
			Step 16: The API returns an "HTTP 200 OK" response with the message "Password reset successful.". The frontend receives the response and redirects the user to the login component
Alternate courses:	**[ALT] Step 5:** If the provided email address does not exist in the database, the system intentionally skips code generation and database insertion. To strictly prevent malicious actors from performing email enumeration, the API still returns the exact same "HTTP 200 OK" response ("If the email exists, a code has been sent."), making failures indistinguishable from successes on the frontend.		
	[ALT] Step 12: The cryptographic comparison determines that the provided code is incorrect. The system increments the "FailedAttempts" integer property on the "PasswordResetCode" entity and commits the change. If "FailedAttempts" reaches or exceeds 5, the system permanently locks the code and returns an "HTTP 401 Unauthorized".		
Conclusion:	The user's account password is securely overwritten with a new hash, and the user regains access to the system.		
Post-condition:	The previous password hash is irrecoverably replaced in the database. The specific 6-digit reset code is marked as consumed.		
Business Rules:	• Secure Access		
Implementation constraints and specifications:	• None		
Assumptions:	• The email provider (Resend) achieves rapid delivery.		
Open issues:	• None		

Table 3: Reset Password - Technical Use Case Narrative

2.1.4 Create User

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 30 April 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create User</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.4</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• Admin</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user signs up and creates a profile. The user will select to sign up for an account and enter their details into the relevant fields. Once done, the system will allow the user to login. On the first login, the user will complete the onboarding details screens. The use case ends once the new user is registered on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User does not already have an account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to create an account.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to create an account.</td><td></td><td></td></tr>
<tr><td>Step 2: The user clicks the “Sign up here” link.</td><td></td><td>Step 3: The system loads and displays the Sign Up form using the Angular frontend with the following controls:
• A Sign up title.
• A Already have an account?</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			Sign in here label. • A First name label and input field. • A Surname label and input field. • A Date of birth label and date picker. • Email label and input field. • Password label and input field. • Confirm password label and input field. • Sign up button.
	Step 4: The user enters their details and clicks the "Sign up" button. [ALT]		Step 5: The system validates the details entered, validating the following: • All fields are completed. • The email is not already in use. • The date of birth is not in the future. • Password and Confirm password match. [ALT]
			Step 6: The Angular frontend calls the Auth service to send a POST request to the C# Web

INTEGRATED NGO MANAGEMENT SYSTEM			
			API backend Auth controller through the Register endpoint. The backend creates a new user in the following entities: User Entity: • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserEmail • PasswordHash • ActiveStatus [ALT]
			Step 7: The system invokes use case 1.1 (User Login) before completing the onboarding details screens.
	Step 8: The user enters their login details.		Step 9: The system loads and displays the Onboarding Details form using the Angular frontend with the following controls: • A Complete your profile title. • A few quick details and you're in label. • A circle with a user icon and the text, Select Profile Photo.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Click the circle to upload a profile picture (optional) label. • A Shirt size label. • Buttons for sizes: ○ XS ○ S ○ M ○ L ○ XL • A Continue button.
	Step 10: The user provides the details and clicks “Continue”.		Step 11: The system validates the details entered, validating the following: • All required fields are completed. [ALT]
			Step 12: The Angular frontend calls the User service to send a POST request to the C# Web API backend User controller through the CompleteOnboarding endpoint. The backend adds the following details to the user in the following entities: User Entity: • ProfilePicture • UserShirtSize

INTEGRATED NGO MANAGEMENT SYSTEM			
			[ALT]
			Step 13: The system loads and displays the Onboarding Skills form using the Angular frontend with the following controls: • A Select your skills title. • A Pick as many as you like – or none at all label. • A search bar with a Search skills placeholder. • A Clear button. • A table of the different skills with a short description for each skill. • A Selected label with a counter for how many were selected. • A Continue button. The skills table is populated with skills retrieved from the Skill table. The Angular frontend calls the Skills service to send a GET request to the C# Web API backend Skills

INTEGRATED NGO MANAGEMENT SYSTEM			
			controller through the GetSkills endpoint. Skill Entity: • SkillName • SkillDescription
	Step 14: The user selects their skills can clicks “Continue”.		Step 15: The system validates the skills, validating the following: • At least one skill selected. [ALT]
			Step 16: The Angular frontend calls the User service to send a POST request to the C# Web API backend User controller through the SaveMySkills endpoint. The backend adds the following details to the user’s account in the following entities: UserSkill Entity: • UserId (PK, FK) • SkillId (PK, FK) [ALT]
			Step 17: The system loads and displays the Onboarding Status form using the Angular frontend with the following controls: • A One last thing title.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A What's your availability right now label. • A User Status label. • A dropdown with a Select a status placeholder. • A Finish button.
	Step 18: The user selects their status and clicks "Finish".		Step 19: The system validates the entered details. The system validates the following: • An option has been selected from the dropdown. [ALT]
			Step 20: The Angular frontend calls the User service to send a POST request to the C# Web API backend User controller through the SaveMyStatus endpoint. The backend adds the following details to the user's account in the following entities: User Entity: • UserStatus [ALT]
			Step 21: The system will then record the transaction in the AuditLog table to

INTEGRATED NGO MANAGEMENT SYSTEM			
			maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the new user. The system will store information such as the authentication user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a new user has been created. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
Alternate courses:	[ALT] Step 4: The user selects the "Already have an account? Sign in here" link. Invoke use case 1.1 (User Login).		
	[ALT] Step 5: The validation fails. The system displays an error message. Go back to step 4.		
	[ALT] Step 6: The system fails to add the new user's details. The system will display an error message based on the status code.		
	[ALT] Step 11: The validation fails. The system displays an error message. Go back to step 10.		
	[ALT] Step 12: The system fails to add the new users onboarding details. The system will display an error message based on the status code.		

INTEGRATED NGO MANAGEMENT SYSTEM	
	[ALT] Step 15: The validation fails. The system displays an error message. Go back to step 14.
	[ALT] Step 16: The system fails to add the new users onboarding skills. The system will display an error message based on the status code.
	[ALT] Step 19: The validation fails. The system displays an error message. Go back to step 18.
	[ALT] Step 20: The system fails to add the new users onboarding status. The system will displays an error message based on the status code.
Conclusion:	The user is successfully captured on the system.
Post-condition:	• The user's profile is created on the system. • The user can be assigned to projects.
Business Rules:	• Valid email address is required.
Implementation constraints and specifications:	• Passwords are hashed before storage in the database.
Assumptions:	• None
Open issues:	• None

Table 4: Create User - Technical Use Case Narrative

2.1.5 Update User

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 30 April 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Update User</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.5</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• Admin</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user updates their profile. The user will navigate to the profile icon in the navbar and select profile. The system will display the user's current details on the system and allow the user to update their details. The user will update the details they wish to and then select save. The system will validate the details entered before updating the user's details on the system. The use case ends once the user's details are successfully updated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User is logged into the system and on their account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to update their details on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to update their details on the system.</td><td></td><td>Step 2: The system invokes use case 1.6 (View User).</td></tr>
<tr><td>Step 3: The user clicks the "Update Profile" button.</td><td></td><td>Step 4: The system loads and displays the Update Profile modal using the Angular frontend with the following controls</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			populated with the users current details: • An Update Profile title. • A Profile Picture label. • A section displaying selectable avatar images. • A circular upload option with an UPLOAD label for adding a custom profile picture. A form divided into two columns: • Left column: • A Name label with an input field. • An Email label with an input field. • Phone Number label with an input field. • A Shirt Size label with a dropdown selection (XS, S, M, L, XL). • Right column:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Surname label with an input field. • A Date of Birth label with a date picker. • A Status label with a dropdown selection. • Session Length label with a dropdown selection • A Skills label. • A dropdown to select a skill. • An Add button to add the selected skill. • A list of added skills displayed as removable pills (each with an "×" remove option). • A Save button. • A Cancel button.
	Step 5: The user updates their details and clicks the "Save" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All fields are completed.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Date of birth is not in the future. • Email does not already exist. [ALT]
			Step 7: The Angular frontend calls the User service to send a PUT request to the C# Web API backend User controller through the UpdateUser endpoint. The backend updates the user details in the following entities: User Entity: • UserName • UserSurname • UserDateOfBirth • UserEmail • UserShirtSize • UserStatus • SessionDurationMinutes UserSkill Entity: • UserId (PK, FK) • SkillId (PK, FK) [ALT]
			Step 8: The system will then record the transaction in the AuditLog table to maintain a history of actions performed

INTEGRATED NGO MANAGEMENT SYSTEM			
			within the system. This is handled in the backend after the successful update of the user. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a user has been updating. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a success toaster informing the user that their details were updated successfully.
Alternate courses:	[ALT] Step 5: The user clicks the "Cancel" button. Use case terminates.		
	[ALT] Step 6: The validation fails. The system displays an error message. Go back to step 5.		
	[ALT] Step 7: The system fails to update user's details. The system will display an error message based on the status code.		
Conclusion:	The user's updated details are successfully captured on the system.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Post-condition:	The user's profile is updated on the system.
Business Rules:	• None
Implementation constraints and specifications:	• The user requires an Internet connection to access this feature.
Assumptions:	• None
Open issues:	• None

Table 5: Update User - Technical Use Case Narrative

2.1.6 View User

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 30 April 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View User</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.6</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user views their profile. The user will navigate to the profile icon in the navbar and select profile. The system will retrieve and display the user’s current details on the system. The use case ends once the user can view their profile.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User is logged into the system and on their account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to view their profile on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to view their profile on the system.</td><td></td><td></td></tr>
<tr><td>Step 2: The user navigates to and selects the profile icon. They then select the “My Profile” option.</td><td></td><td>Step 3: The Angular frontend calls the User service to send a GET request to the C# Web API backend User controller through the GetUserProfileById endpoint. The backend</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			retrieves the user's details from the following entities: User Entity: • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserPhoneNumber • UserEmail • UserStatus • ProfilePicture • UserShirtSize • SessionDurationMinutes UserSkill Entity: • UserId (PK, FK) • SkillId (PK, FK) Skill Entity: • SkillId (PK) • SkillName [ALT]
			Step 4: The system loads and displays the User Profile screen using the Angular frontend with the following controls populated with the user details: A header section containing: • The user's profile image. • A display of the user's full name (name and surname).

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A display of the user's email address. A profile details card containing: • A row showing the Name label and the user's name. • A row showing the Surname label and the user's surname. • A row showing the Email label and the user's email. • A row showing the Phone number label and the user's phone number. • A row showing the Date of birth label and the user's date of birth. • A row showing the Shirt size label and the user's selected shirt size. • A row showing the Status label and the user's current status. • A row showing the Session length label and the user's current session length.

INTEGRATED NGO MANAGEMENT SYSTEM			
			A My skills section displaying the user's skills as individual skill pills. A fallback message indicating "No skills added yet" if no skills exist. A set of profile actions: • A Update Profile button • A button to toggle account status, displaying either Deactivate Account or Reactivate Account depending on the current state.
Alternate courses:	[ALT] Step 3: The system fails to retrieve the user's details. The system will display an error message based on the status code.		
Conclusion:	The user can view their profile.		
Post-condition:	The user's profile is displayed.		
Business Rules:	• The user has an account on the system.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 6: View User - Technical Use Case Narrative

2.1.7 Reactivate User

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 30 April 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Reactivate User</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.7</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user reactivates their account. The user will login into their account before being asked if they would like to reactivate their account. Once reactivated the system will display a success popup. The use case ends once the users account is successfully reactivated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User’s account is deactivated.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to reactivate their profile on the system.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to reactivate their profile on the system.</td><td></td><td>Step 2: The system invokes use case 1.1 (User Login).</td></tr>
<tr><td>Step 3: The user enters their login details.</td><td></td><td>Step 4: The system loads and displays the Reactivate Account screen using the Angular frontend with the following controls:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			• An icon inside a circular container. • An Account Deactivated title. • A label indicating that the account linked to the user's email address has been deactivated, asking whether the user would like to reactivate it. • A Yes, reactivate my account button • A No, go back to login button
	Step 5: The user clicks the "Yes, reactivate my account" button. [ALT]		Step 6: The Angular frontend calls the Auth service to send a POST request to the C# Web API backend Auth controller through the ReactivateAccount endpoint. The backend sets the user's active status to true (1) in the following entities: User Entity: • AcitveStatus [ALT]

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful reactivation of the user. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a user has been reactivated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system loads and displays the Welcome Back screen using the Angular frontend with the following controls: • An icon inside a circular container

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Welcome Back! title. • A label confirming that the account has been successfully reactivated. • A label indicating that the user will be redirected automatically.
Alternate courses:	[ALT] Step 5: The user clicks No, go back to login. The user is then redirected back to the login screen.		
	[ALT] Step 6: The system fails to update the user's active status. The system will display an error message based on the status code.		
Conclusion:	The system displays a welcome back popup informing the user that their account has been successfully reactivated.		
Post-condition:	• The user's account is reactivated on the system. • User can be assigned to projects.		
Business Rules:	• Admins can reactivate users' accounts.		
Implementation constraints and specifications:	• The user requires an Internet connection to access this feature.		
Assumptions:	• None		
Open issues:	• None		

Table 7:Reactivate User - Technical Use Case Narrative

2.1.8 Deactivate User

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 30 April 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate User</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">1.8</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user deactivates their account. The user will navigate to the profile icon in the navbar and select profile. The system will display the user's current details on the system and allow the user to deactivate their account. Once deactivate the user is logged out of the system. The use case ends once the users account is successfully deactivated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User is logged into the system and on their account.
User's account is active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to deactivate their profile on the system.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to deactivate their profile on the system.</td><td></td><td></td></tr>
<tr><td>Step 2: The user click the profile icon and then selects the "My profile" option.</td><td></td><td>Step 3: The system invokes use case 1.6 (View User).</td></tr>
<tr><td>Step 4: The user clicks the "Deactivate Account" button.</td><td></td><td>Step 5: The system displays a confirmation</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			modal with the following controls: • Confirm Deactivation title. • A Are you sure you want to deactivate your account label. • A Confirm button • A Cancel button
	Step 6: The user clicks “Confirm”. [ALT]		Step 7: The Angular frontend calls the User service to send a PUT request to the C# Web API backend User controller through the DeactivateUser endpoint. The backend sets the user’s active status to false (0) and user status to “Not available for projects” in the following entities: User: • AcitveStatus • UserStatus [ALT]
			Step 8: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the

INTEGRATED NGO MANAGEMENT SYSTEM			
			backend after the successful deactivation of the user. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a user has been deactivated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system invokes use case 1.2 (User Logout).
Alternate courses:	[ALT] Step 6: The user clicks “Cancel”. Use case terminates.		
	[ALT] Step 7: The system fails to update the user’s active status. The system will display an error toaster based on the status code.		
Conclusion:	The user’s account is deactivated.		
Post-condition:	• The user’s account is deactivated on the system. • User cannot be assigned to projects.		
Business Rules:	• None		
Implementation constraints and specifications:	• The user requires an Internet connection to access this feature.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Assumptions:	• None
Open issues:	• None

Table 8: Deactivate User - Technical Use Case Narrative

2.1.9 View FAQ

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 17 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View FAQs</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>1.9</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin views the platform's frequently asked questions. After navigating to the FAQs page, the system retrieves all FAQ categories and FAQ entries and displays them as a collapsible accordion list. The Admin can narrow the list by typing a keyword or selecting a category and can expand any entry to read its full answer along with the audit details recorded against it. No data is created, updated, or deleted.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated and at least one FAQ category exists in the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin navigates to the FAQs page, either by entering the URL directly or by selecting the FAQs card on the Dashboard.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: Admin navigates to the FAQs page.</td><td></td><td>Step 2: The system loads the FAQs screen and calls “GET api/faqcategory/categories”, which retrieves all FAQ categories from the FAQCategory entity in alphabetical order. The categories populate the filter</td></tr>
</table>

			dropdown, with "All" selected by default. [ALT]
			Step 3: The system calls "GET api/faq", which retrieves every FAQ by joining the FAQ and FAQCategory entities. No filtering, sorting, or paging is applied to the query. [ALT]
			Step 4: The system displays the FAQs page with a search field, a category dropdown, and the accordion list. Each collapsed entry shows a category badge and the question text.
	Step 5: Admin types a keyword into the search field.		**Step 6:** The system filters the list to FAQs whose question or answer contains the keyword and re-renders the accordion. Filtering is performed on the records already retrieved, so no further request is made. [ALT]
	Step 7: Admin selects a category from the dropdown. [ALT]		**Step 8:** The system filters the list to FAQs belonging to the selected category. If a keyword has also been entered, both conditions are applied together. [ALT]
	Step 9: Admin selects an FAQ entry.		**Step 10:** The system expands the entry to show the full answer and an audit footer recording the user who created the FAQ and the date it was created. Selecting the entry again collapses it.
Alternate courses:	**[ALT] Step 2:**		

	The FAQ categories cannot be retrieved. The system displays "Failed to load FAQ categories." and the category dropdown remains empty.
	[ALT] Step 3: The FAQs cannot be retrieved. The system displays "Failed to load FAQs." and the accordion list remains empty.
	[ALT] Step 7: Admin selects "All" from the dropdown. The category condition is bypassed and FAQs from every category are shown, subject to any keyword entered.
	[ALT] Step 6/8: No FAQs match the keyword or the selected category. The system displays "No FAQs match your filters." in place of the accordion list, and the search field and dropdown remain available.
Conclusion:	The use case concludes when the FAQ list is displayed to the Admin, filtered as required, with the selected entries expanded.
Post-condition:	No data is modified. The keyword and category selections remain in effect until the Admin navigates away from the page.
Business rules:	• Only authenticated users can view the FAQs page. • Search is case-insensitive and is applied to both the question and the answer text. • A keyword search and a category selection are applied together when both are active. • Categories are listed alphabetically by name, and FAQs are listed in the order they were created. • Multiple FAQ entries may be expanded at the same time.
Implementation constraints and specifications:	• "FAQ entries are keyboard accessible and expose their expanded state to screen readers.
Assumptions:	• The default FAQ categories are present in the database after installation. • The Admin's browser supports the styling and scripting features used by the page.
Open issues:	• Searching and filtering are performed on the retrieved records, so the entire FAQ table is loaded on every visit.

Table 9: View FAQ - Technical Use Case Narrative

2.2 Admin Subsystem Use Case Narratives

2.2.1 Create User Profile

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 1 May 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create User Profile</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">2.1</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• User (ERA)</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin creates a new user profile. The system will display all the users on the system and allow the super admin to create a new user profile. The super admin will enter the user’s details and select create. The use case ends once the user’s profile is successfully saved on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system and on their account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to create a user profile.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to create a user profile.</td><td></td><td>Step 2: The system invokes use case 2.3 (View User Profile)</td></tr>
<tr><td>Step 3: The super admin clicks the “Create” button</td><td></td><td>Step 4: The system loads and displays the Create User modal using the Angular frontend with the following controls:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Create User Profile title. • A Name label and input field. • A Surname label and input field. • An Email label and input field. • A Date of Birth label and date picker. • A Phone Number label and input field. • A Create button. • A Cancel button.
	Step 5: The super admin enters the user's details and clicks the "Create" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All fields are completed. • The email is not already in use. • The date of birth is not in the future. • Password and Confirm password match. [ALT]
			Step 7: The Angular frontend calls the User service to send a POST request to the C# Web

INTEGRATED NGO MANAGEMENT SYSTEM			
			API backend User controller through the CreatUser endpoint. The backend creates a new user in the following entities: User Entity: • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserEmail • PasswordHash • ActiveStatus [ALT]
			Step 8: The system sends an email via the email service in the backend to the user indicating their account has been created as well as a temporary password.
			Step 9: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the user. The system will store information such as the

INTEGRATED NGO MANAGEMENT SYSTEM			
			authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a user has been created. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 10: The system displays a success toaster informing the super admin that the new user was created successfully.
Alternate courses:	[ALT] Step 5: The super admin clicks the Cancel button. The super admin is redirected to the User Management screen.		
	[ALT] Step 6: The validation fails. The system displays an error message. Go back to step 5.		
	[ALT] Step 7: The system fails to add the new user's details. The system will display an error message based on the status code.		
Conclusion:	The user is successfully created on the system.		
Post-condition:	The user can login and complete the onboarding steps.		
Business Rules:	• Only super admins can create user profiles.		
Implementation constraints and specifications:	• Only super admins or user's with the required permissions can create user profiles.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Assumptions:	• Super admin has the correct user details.
Open issues:	• None

Table 10: Create User Profile - Technical Use Case Narrative

2.2.2 Update User Profile

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Christopher Bailey		**Date: 1 May 2026** **Version: 1**
USE CASE NAME:	Update User Profile		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	2.2		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	• None		
Description:	This use case describes the event in which a super admin updates a user's profile. The system will display all the users on the system and allow the super admin to select a user profile. The system will display the user's current details and allow the user to update their details. The use case ends once the user's profile is successfully updated on the system.		
Pre-condition:	Super Admin is logged into the system and on their account.		
Trigger:	The super admin requests to update a user profile.		
Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: The super admin requests to create a user profile.		Step 2: The system invokes use case 2.3 (View User Profile)
	Step 3: The super admin selects "Update" on a user.		Step 4: The system loads and displays the Update User Profile modal using the Angular frontend with the following controls: • A header section

INTEGRATED NGO MANAGEMENT SYSTEM			
			containing an Update title. • A circular avatar displaying the user's current profile picture. A form divided into two columns. Left column: • A Name label with an input field. • An Email label with an input field. • A Phone Number label with an input field. • A Shirt Size label with a dropdown selection (XS, S, M, L, XL). Right column: • A Surname label with an input field. • A Date of Birth label with a date picker. • A Status label with a dropdown selection.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Save button to submit the changes. • A Cancel button to close the modal.
	Step 5: The super admin enters the updated user's details and clicks the "Save" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All fields are completed. • Date of birth is not in the future. • Email does not already exist. [ALT]
			Step 7: The Angular frontend calls the User service to send a PUT request to the C# Web API backend User controller through the UpdateUser endpoint. The backend updates the user details in the following entities: User • UserName • UserSurname • UserDateOfBirth • UserEmail • UserShirtSize • UserStatus

INTEGRATED NGO MANAGEMENT SYSTEM			
			[ALT]
			Step 8: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the user. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a user has been updated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a success toaster informing the super admin that the user was updated successfully.
Alternate courses:	[ALT] Step 5: The super admin clicks the Cancel button. Use case terminates.		
	[ALT] Step 6:		

INTEGRATED NGO MANAGEMENT SYSTEM	
	The validation fails. The system displays an error message. Go back to step 5.
	[ALT] Step 7: The system fails to update the user's details. The system will display an error message based on the status code.
Conclusion:	The system displays a success toaster informing the super admin that the user was updated successfully.
Post-condition:	The user's details are updated.
Business Rules:	• Only super admins or user's with the required permissions can update user profiles.
Implementation constraints and specifications:	• The user requires an Internet connection to access this feature.
Assumptions:	• Super admin has the correct updated user details.
Open issues:	• None

Table 11: Update User Profile - Technical Use Case Narrative

2.2.3 View User Profile

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Nicholas Bailey	**Date: 30/04/2026** **Version:** 1	
USE CASE NAME:	View User Profile	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	2.3		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	• None		
Description:	This use case describes when an admin views the details of all the users within the system. The system displays the skills in a table containing the user's name, surname, email, date of birth, shirt size, and status. The use case ends when the admin sees all the users and their details.		
Pre-condition:	• The admin must be logged into the system.		
Trigger:	The admin wants to view the users on the system.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The admin wants to view the users on the system.		
	Step 2: The admin hovers the "Admin" tab and then selects "Users". [ALT]		**Step 3:** The system sends the request through the Angular frontend and user service to the relevant HTTP GET API endpoint in Visual Studio (UserController – GetAllUsers), which retrieves all users from the User table in the database.

INTEGRATED NGO MANAGEMENT SYSTEM			
			User Entity: • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserEmail • UserStatus • UserShirtSize
			Step 4: The system will then display the User Management page using the Angular frontend, showing all retrieved users in a table based on the data returned from the API. • Page header section: ○ Page title (“User Management”) ○ Description text (“Create, edit, and manage users.”) ○ Create user button (Top right of table.) • Table display section: ○ Users are displayed in a table • The table includes the following columns: ○ Name ○ Surname ○ Email ○ Date of Birth

INTEGRATED NGO MANAGEMENT SYSTEM			
			○ Shirt Size (displayed as a styled tag) ○ Status (displays current user status or "N/A" if not available) ○ Actions • Actions column: ○ Update button (opens the update functionality for the selected user) ○ Deactivate/Reactivate button: ▪ Displays "Deactivate" if the user is active ▪ Displays "Reactivate" if the user is inactive ▪ Button styling changes dynamically based on user status [ALT]
	Step 5: The admin can view the users.		
Alternate courses:	**ALT-Step 2:** User is on mobile and selects the menu icon. Go to step 3.		
	ALT-Step 4: The system does not find any users. Terminate use case.		
Conclusion:	All users' details are displayed in a table.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Post-condition:	The system displays all the current users in a table.
Business Rules:	• Only super admins or users given appropriate permissions can view user's profiles.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 12: View User Profile - Technical Use Case Narrative

2.2.4 Deactivate User Profile

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate User Profile</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">2.4</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin deactivates a user in the system. The system allows the Super Admin to select deactivate on a user and will then confirm the deactivation. Once confirmed, the system updates the user’s status to inactive in the database. The use case ends when the user’s status is changed to inactive.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to deactivate a user on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to deactivate a user on the system.</td><td></td><td>Step 2: The system invokes UC 2.3 (View User Profile) to display the users in a table. This is done using the Angular frontend and users service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (UserController).</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	Step 3: The super admin selects the "Deactivate" button on a specific user.		**Step 4:** The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Deactivate" button. • Confirmation modal title ("Confirm user deactivation") • Confirmation message label ○ Displays message: "Are you sure you want to deactivate this user?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system will then deactivate the selected user by routing the action through the Angular frontend and user service, which sends the request to the appropriate HTTP DELETE endpoint in Visual Studio (UserController – DeactivateUser). The API endpoint will validate the user and

INTEGRATED NGO MANAGEMENT SYSTEM			
			update its status in the database using Entity Framework Core in MS SQL Server. This ensures that the user is soft deleted. The following entity is involved in the process: **User Entity:** • UserId (PK) • UserName • UserSurname • UserDateOfBirth • UserEmail • UserPassword • ProfilePicture • ActiveStatus • UserStatus • UserShirtSize [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the user. The system will store information such as the authenticated user performing the action, the timestamp of the

INTEGRATED NGO MANAGEMENT SYSTEM			
			operation, and a descriptive log entry indicating that a user has been deactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system will then display a confirmation message indicating that the user has been successfully deactivated. This is implemented in the Angular frontend using a toaster notification component. The confirmation message will display the text: "User deactivated successfully".
Alternate courses:	**ALT-Step 5:** The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 2.		
	ALT-Step 6: The system fails to deactivate the skill and displays a toaster message: "Failed to deactivate user". Terminate use case.		
Conclusion:	The active user has been deactivated, and a confirmation message is shown.		
Post-condition:	The system updates the user's status to inactive in the User table.		
Business Rules:	• Only super admin or users given appropriate permissions can deactivate a user.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		

INTEGRATED NGO MANAGEMENT SYSTEM	
Open issues:	• None

Table 13: Deactivate User Profile – Technical Use Case Narrative

2.2.5 Add FAQ

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Add FAQ	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.5		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process where a Super Admin establishes a new Frequently Asked Question entry within the system directory. The operator inputs a targeted category, question, and rich-text solution statement into an Angular Reactive Form layout. Upon clicking submit, the client interface checks form validation criteria and dispatches an asynchronous JSON payload to the corresponding backend .NET 8 API router (POST api/Faq/add). The backend verifies text string properties against duplicate entries before leveraging Entity Framework Core to write the record to the relational database tables. The use case ends when the frontend updates local state array caches, dismisses the editing overlay window, and renders a green success confirmation toast.		
Pre-condition:	• The Super Admin must hold appropriate system configuration management permissions, and at least one functional FAQ classification type category must exist in the database partitions.		
Trigger:	The Super Admin clicks the "Create FAQ" action button trigger element layout component.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to create a new FAQ by clicking the "Create		**Step 2:** The Angular Router intercepts the route request and initializes the FAQ

	FAQ" action trigger button.		Creation Component. The framework processes the HTML layout context and updates the DOM elements.
		Step 3: The system displays the user interface window presenting an open Angular Reactive Form Group layout containing an unpopulated category select dropdown array, an empty text input field for the question, and a rich text textarea field for the answer details.	
	Step 4: The Super Admin selects the relevant category, inputs the question text content, captures the detailed answer content, and clicks the primary "Commit Records" submit button. [ALT]		**Step 5:** The client-side application captures the structural data state. The Angular FAQService constructs and dispatches an asynchronous HTTP POST request to the backend API endpoint string api/Faq/add enclosing an AddFaqRequest JSON payload container. The .NET 8 API FaqController intercepts

			the incoming network payload stream. The backend runtime references the assigned ApplicationDbContext instance and fires an asynchronous verification lookup query against the SQL table partitions via Entity Framework Core to verify that an identical duplicate question is not already saved in that target category mapping.
			Step 6: The backend server engine executes core validation parameters against incoming JSON parameters before authorizing transaction storage updates. The framework performs model validation checks to confirm mandatory rows are present, verify character array length thresholds, map explicit category lookups, and validate Super Admin security credentials.
			Step 7: Upon successful logic validation confirmations, the system writes the new entity record tracking state directly into the DbContext Faq model array collection and

			executes an asynchronous storage save block routine command (SaveChangesAsync)
			Step 8: The API data pipeline transmits an HTTP 201 Created confirmation response block stream back to the calling client. The Angular frontend catches the execution status code, modifies the local application state FAQ cache arrays, shuts down the UI modal overlay display layers, and fires a green success toast alert saying: "FAQ added successfully."
Alternate courses:	**[ALT] Step 4:** The user chooses to cancel the action and clicks the "Discard" button. The frontend interface closes the editing modal panel immediately, drops all form mutations from active memory buffers, and prevents network calls to the server.		
Conclusion:	The use case concludes once all data rules clear validation, records register successfully inside the database backend tables, form inputs clear, and dashboard lookup lists update.		
Post-condition:	• A new FAQ entry is permanently written to the database storage layer, immediately matching it against its target category index for end-user filter selection workflows.		
Business rules:	• Only users with explicitly assigned Super Admin security authorization claims can insert new FAQ entities into global datasets. • The system blocks the generation of any new FAQ record that contains an identical question text payload within the exact same category mapping.		

Implementation constraints and specifications:	• Alphanumeric text data mapping to the question input field parameters must not exceed a size floor ceiling threshold of 250 characters.
Assumptions:	• None
Open issues:	• None

Table 14: Add FAQ - Technical Use Case Narrative

2.2.6 Update FAQ

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Update FAQ	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.6		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process for modifying an existing FAQ entry's category index, question text, or rich-text solution scope details. The user adjusts configuration parameters within an active frontend reactive layout modal. Upon clicking save, the client interface checks field validity maps and dispatches an asynchronous PUT request to the .NET 8 API endpoint (PUT api/Faq/update/{id}). The backend locates the target entity row via Entity Framework Core tracking contexts and overwrites its structural schema properties with the new values. The use case finishes once mutations write safely to disk, modal popup blocks close, and lookup views across the application refresh.		
Pre-condition:	• The targeted FAQ entry record must exist in the database partition, and the user must hold necessary administrative configuration write claims.		
Trigger:	The user clicks the context "Edit" button layout element located next to a specific FAQ data row line item.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to edit an existing FAQ entry row item by hitting the specific contextual "Edit" action button trigger.		**Step 2:** The frontend application catches the click event and launches the edit workflow modal dialog display framework. The Angular service dispatches an

			automated background HTTP GET request to retrieve the latest backend database metrics.
		Step 3: The UI component parses the incoming data model values and executes an explicit reactive form patch configuration override command (patchValue) to populate the text inputs automatically with current record attributes for user modification.	
	Step 4: The Super Admin navigates across the active form properties, updates the targeted description inputs or category mapping lists, and hits the "Save Changes" execution control button. [ALT]		**Step 5:** The client framework assesses form validity parameters on the client layer. The Angular FAQService packages the modifications and sends an HTTP PUT request to the route path index api/Faq/update/{id} wrapping the UpdateFaqRequest body details. The .NET API controller intercepts the transmission stream,

			activates its ApplicationDbContext tracking instance, and invokes an asynchronous data lookup query statement (SingleOrDefaultAsync) against the "Faq" entity table to locate the specific primary key row index.
			Step 6: The system backend architecture executes model integrity checking algorithms against the incoming payload to verify the record identity exists, assert string boundary lengths, and ensure the session holds active Super Admin authorisation claims [ALT].
			Step 7: Upon validation success, the system maps the payload attributes over the tracked entity data properties using the ORM tracker and signs off database table record modification updates via an asynchronous save update call (SaveChangesAsync).
			Step 8: The web API returns an HTTP 200 OK completion confirmation payload. The Angular frontend catches the

			message code, overrides the stale object instance within its active data array cache, removes the modal view wrapper from the DOM, and fires a green success notification alert box reading "FAQ updated successfully."
Alternate courses:	**[ALT] Step 4:** The user aborts the workflow and clicks the "Discard" or "Cancel" button. The frontend interface closes the editing panel immediately, drops all form mutations from memory buffers, and prevents network data calls to the server.		
	[ALT] Step 6: The data validation layer discovers a naming or question text conflict matching another active entry row or detects that the target index has been deleted from memory. The API stops the process pipeline and returns an error response code (HTTP 409 Conflict). The client application catches the exception tracking details and displays a clear red validation error message banner.		
Conclusion:	The use case concludes when the system successfully validates the updated data parameters, saves the transaction metrics to database storage files, and refreshes the main grid layout view.		
Post-condition:	• The target FAQ instance's database entity fields are permanently updated with the new text values, ensuring altered parameters reflect across downstream interface views.		
Business rules:	• Only users with explicitly assigned Super Admin configuration privileges can modify system FAQ entries.		
Implementation constraints and specifications:	• Field modifications targeting the primary question string must strictly maintain compliance with the 250-character threshold limit.		
Assumptions:	• None		
Open issues:	• None		

Table 15: Update FAQ - Technical Use Case Narrative

2.2.7 Remove FAQ

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Remove FAQ	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.7		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the structural steps executed when a Super Admin deletes an FAQ entry completely from persistent database systems. The user triggers the action from the row context. The frontend generates an explicit validation warning alert overlay modal. Upon confirmation, the client initiates an asynchronous HTTP DELETE transaction directly to the corresponding backend .NET 8 API controller routing path (DELETE api/Faq/remove/{id}). The backend executes validation constraint assessments before completely dropping the targeted row record index from persistent tables via Entity Framework Core context trackers. The use case concludes when the server commits changes, and the client removes the targeted entry from local data grid layers.		
Pre-condition:	• The targeted FAQ tracking record must currently exist inside persistent SQL tables with an active index match.		
Trigger:	The user clicks the specific row deletion contextual action "Delete" icon trigger element layout asset.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to drop an FAQ tracking item line row by clicking its specific row deletion contextual action icon trigger.		

		Step 2: The frontend system halts current layout sequences and displays a modal confirmation warning overlay dialog framework, presenting a descriptive message text to the operator: "Are you sure you want to permanently delete this FAQ item? This action cannot be undone."	
	Step 3: The Super Admin assesses the warning text context and hits the primary confirmation "Confirm Delete" action switch inside the modal overlay. [ALT]		**Step 4:** The frontend interface captures the user click intent. The Angular FAQService formats an asynchronous HTTP DELETE request command routing directly to the backend endpoint URL parameter path map api/Faq/remove/{id}. The backend web API intercepts the data deletion stream request. The data runtime environment establishes connections to the ApplicationDbContext unit tracking context and runs an execution query

			sweep targeting the row index matching the supplied ID property.
			Step 5: The backend system subjects the command parameters to server constraints to verify database row existence and match current user claims data models against required administrative permission sets.
			Step 6: The server marks the identified entity row record for deletion within the data set tracker context and permanently drops the row record from persistent storage tables via an asynchronous change statement execution (SaveChangesAsync).
			Step 7: The API transfers an HTTP 200 OK or HTTP 240 No Content message tracking packet block. The client application interceptors catch the response, splice the target item index out of its local collection layout arrays, dismiss the alert modal wrapper layer, and render a green toast confirmation message: "FAQ deleted successfully."
Alternate courses:	**[ALT] Step 3:**		

	The user chooses to cancel the action and clicks the "Cancel" button within the warning dialog block layout. The confirmation modal overlay closes immediately, the workflow cancels, and the FAQ's database status remains completely unchanged.
Conclusion:	The use case concludes once all data rules clear validation, records drop successfully from database tables, the confirmation dialog closes, and dashboard views refresh to reflect the deletion.
Post-condition:	• The targeted FAQ entry row is permanently dropped from persistent relational database storage maps, rendering the tracking parameters unrecoverable.
Business rules:	• Hard-deletion commands targeting FAQ database items are restricted strictly to users with active Super Admin security claims to safeguard data dictionary integrity.
Implementation constraints and specifications:	• The operation must execute sequentially over an asynchronous pipeline to prevent UI blocking during processing loops.
Assumptions:	• None
Open issues:	• None

Table 16: Remove FAQ - Technical Use Case Narrative

2.2.8 Add FAQ Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 24 June 2026 **Version: 1.0**	
Use case name:	Add FAQ Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.8		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case describes the technical steps required to establish a new category label within the FAQ classification lookups system. The user inputs category metrics into an Angular Reactive Form layout overlay modal. Upon clicking submit, the frontend checks form control validation syntax states and dispatches a JSON payload to the corresponding .NET 8 API backend endpoint (POST api/FaqType/create) packing a structured CreateFaqTypeRequest body. The backend processes the incoming package payload context and uses Entity Framework Core to write the new classification record to the relational database tables. The use case finishes once the frontend updates local state lookup arrays, clears active modal containers, and triggers a green confirmation toast notification.		
Pre-condition:	• The executing user must hold appropriate system configuration management privileges, and the system must be deployed ahead of testing cycles.		
Trigger:	The user clicks the "Create FAQ Type" action button.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to create a new FAQ category grouping by clicking the "Create FAQ		

	Type" action trigger button.		
		Step 2: The system launches the category generation modal layer overlay. The frontend generates and renders an open Angular Reactive Form Group interface layout containing an alphanumeric text input field for the FAQ Type Name, a text area for the Description Scope, an option selector for status, a "Discard" button, and a primary "Commit Records" submit button.	
	Step 3: The Super Admin populates the category type name input field, enters descriptive details about the scope, selects an initial active configuration state, and		**Step 4:** The client application evaluates form field control validation states. The Angular FAQService packages the data components and dispatches an

	clicks the primary "Commit Records" button. [ALT]		asynchronous HTTP POST payload statement directly to the API path api/FaqType/create enclosing a CreateFaqTypeRequest JSON body. The backend controller receives the incoming request. The framework links directly into the ApplicationDbContext runtime instance and executes an asynchronous database search scan against the "FaqType" entity table to verify that an identical category name string does not already exist.
			Step 5: The server backend evaluates data input constraints before authorizing database persistence. The framework runs validations to confirm all mandatory rows are present, verify text string lengths stay within schema boundary structures, and confirm the active session tokens contain Super Admin operational claim permissions. FaqType Entity Schema Structure: • FaqTypeId (PK) • TypeName

			• DescriptionScope • IsActiveStatus [ALT]
			Step 6: Upon successful validation passes across all system check loops, the system initializes a new instance tracking entry of the FaqType database model table collection and saves the transaction metrics to disk via an asynchronous database commit command (SaveChangesAsync).
			Step 7: The system endpoint returns an HTTP 201 Created code output tracking confirmation parameters. The frontend catches the response message, appends the newly returned category model instance into its master application view display arrays, clears the modal state layer, and triggers a green confirmation toast statement reading: "New FAQ type category successfully added to global system directories."
Alternate courses:	**[ALT] Step 3:** The user chooses to cancel the action and clicks the "Discard" button. The frontend interface closes the editing modal panel immediately, drops all form mutations from memory buffers, and prevents network calls to the server.		

	[ALT] Step 5: The backend service layer triggers a business rule conflict check identifying that an identical classification type name already exists in the table partitions. The system immediately interrupts transaction loops and sends back an HTTP 409 Conflict response code. The frontend layout catches the failure tracking information and renders a prominent red descriptive validation warning layout banner across the modal frame.
Conclusion:	The use case concludes once all data rules clear validation parameters, records register successfully inside database backend tables, and configuration lookup dashboards refresh.
Post-condition:	• A new FAQ category classification option is permanently written to the database, making it immediately available for routing and FAQ entry classification workflows.
Business rules:	• Only users with explicitly assigned Super Admin configuration privileges can create new FAQ lookup categories.
Implementation constraints and specifications:	• FAQ Type Entity Schema Structure: FaqTypeId (PK), TypeName, DescriptionScope, IsActiveStatus.
Assumptions:	• None
Open issues:	• None

Table 17: Add FAQ Type - Technical Use Case Narrative

2.2.9 Update FAQ Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 24 June 2026 **Version: 1.0**	
Use case name:	Update FAQ Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.9		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process for modifying an existing FAQ type classification category's structural name and scope parameters. The user adjusts field metrics inside the active frontend editing modal layout panel. Upon clicking save, the client interface assesses client field validation syntax maps and dispatches an asynchronous PUT call directly to the backend endpoint URL parameter path (PUT api/FaqType/update/{id}). The API endpoint binds to the shared ApplicationDbContext instance and runs an asynchronous lookup transaction seeking a matching primary key sequence. The backend overwrites the tracked database entity schema properties with the new payload values and commits state updates via Entity Framework Core. The use case concludes when changes are written to disk, modal containers close, and configuration dashboards refresh.		
Pre-condition:	• The target FAQ type category record must exist inside the database tables, and the operator must hold valid configuration management write claims.		
Trigger:	The user clicks the row "Edit" button layout element asset next to a specific FAQ type listing line item.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to alter an FAQ category name or description by		

	clicking the row "Edit" button layout element next to a specific FAQ type listing row.		
		Step 2: The frontend triggers the edit workflow modal overlay window container. The system passes the active row parameters and patches the existing database field metrics directly into the form text inputs via an explicit Angular form patch operation (patchValue).	
	Step 3: The Super Admin modifies the alphanumeric text input values (updating category names or adjusting description scope explanations) and clicks the "Save Changes" execution control button. [ALT]		**Step 4:** The client framework verifies form control validity states on the browser layer. The Angular FAQService dispatches a structured HTTP PUT call directly to the backend endpoint URL parameter path api/FaqType/update/{id} passing the comprehensive updated parameter payload maps. The API endpoint processes data extraction streams. The tracking

			logic binds to the shared ApplicationDbContext and runs an asynchronous lookup transaction statement (SingleOrDefaultAsync) seeking a matching FAQ type primary key sequence mapping.
			Step 5: The system evaluates payload modifications against backend validation constraints. It checks that the target FAQ type record is found in the database layer, assures alphanumeric character arrays do not break size properties, and validates that the active session holds necessary configuration management permissions.
			Step 6: Upon completing verification loops, the system overwrites the tracked database entity schema properties with the new payload values and saves changes to the database storage layer via an asynchronous update command (SaveChangesAsync)
			Step 7: The server backend dispatches an HTTP 200 OK status packet back to the client

			interface. The frontend application intercepts the success code message, closes the modal popup block container, updates its local state arrays, and triggers a confirmation alert toast reading "Changes committed successfully."
Alternate courses:	**[ALT] Step 3:** The user aborts the workflow and clicks the "Discard" or "Cancel" button. The frontend interface closes the editing panel immediately, drops all form mutations from memory buffers, and prevents network data calls to the server.		
Conclusion:	The use case concludes when the system successfully validates the modified parameter maps, saves updates to the database storage layer, and refreshes the main dashboard grid lookup views.		
Post-condition:	• The target FAQ type category's database entity properties are permanently updated with the new input text parameters.		
Business rules:	• Only users with explicitly assigned system configuration privileges can modify FAQ classification lookup categories		
Implementation constraints and specifications:	• Configuration mutations must execute over an asynchronous pipeline leveraging Entity Framework tracking contexts.		
Assumptions:	• None		
Open issues:	• None		

Table 18: Update FAQ Type - Technical Use Case Narrative

2.2.10 Remove FAQ Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 24 June 2026 **Version: 1.0**	
Use case name:	Remove FAQ Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	2.10		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process invoked when an authorized user permanently deletes an FAQ category type classification that is no longer needed across systemic directories. The user triggers the action from the lookup grid row. The frontend halts execution flows and presents an explicit modal confirmation warning alert overlay block dialog framework. Upon confirmation, the client interface routes an asynchronous HTTP DELETE request directly to the backend service controller endpoint path (DELETE api/FaqType/remove/{id}). The backend service controller evaluates database dependency constraints before completely marking the tracked record instance for deletion and signing off changes via Entity Framework Core. The use case finishes once table rows are dropped, the modal closes, and dashboard view panels refresh.		
Pre-condition:	• The targeted FAQ type classification category record must exist inside the database context frames and must be free of active downstream dependencies.		
Trigger:	The user clicks the row deletion contextual action icon "Delete" trigger button element asset on a target category row line item.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin requests to drop an FAQ tracking category line row by		

	clicking its row deletion contextual action icon trigger button.		
		Step 2: The frontend system halts current layout sequences and displays a modal confirmation warning overlay dialog framework, presenting a descriptive message text to the operator: "Are you sure you want to permanently delete this FAQ item? This action cannot be undone."	
	Step 3: The Super Admin assesses the warning text context and hits the primary confirmation "Confirm Delete" action switch inside the modal overlay. [ALT]		**Step 4:** The frontend interface captures the user click intent. The Angular FAQService formats an asynchronous HTTP DELETE request command routing directly to the backend endpoint URL parameter path map api/FaqType/remove/{id}. The backend web API intercepts the data deletion stream request. The data runtime environment establishes

			connections to the ApplicationDbContext unit tracking context and runs an execution query sweep targeting the row index matching the supplied ID property.
			Step 5: The backend system runs validation rules before allowing execution. It verifies that the targeted row record exists in the tracking context and confirms that no active FAQ entries are currently classified under this category ID to protect structural cascading foreign key dependencies.
			Step 6: Upon successful logic validation confirmations, the server completely marks the tracked record instance for deletion via the DbContext entity tracker and signs off the record sweep via an asynchronous storage update execute call (SaveChangesAsync).
			Step 7: The API transfers an HTTP 200 OK or HTTP 240 No Content message tracking packet block. The client application interceptors catch the response, splice the target item index out of its local collection layout

			arrays, dismiss the alert modal wrapper layer, and render a green toast confirmation message: "FAQ type category deleted successfully."
Alternate courses:	**[ALT] Step 3:** The operator clicks the "Cancel" or "Discard Deletion" button control asset inside the confirmation dialog panel overlay wrapper layout. The system closes the window immediately, cancels the operational flow, and leaves the database record completely unchanged.		
Conclusion:	The use case concludes once all data integrity validations clear, the target row record drops from persistent system tables, and configurations dashboards refresh local caches.		
Post-condition:	• The specific category row tracking index is permanently removed from system tables, removing it from selection options across systemic forms.		
Business rules:	• Deactivation and deletion requests are blocked if any active downstream FAQ profile entries depend on the category classification index to protect data mapping integrity.		
Implementation constraints and specifications:	• Relational constraints must strictly block deletion actions unless the dependent array collection size equals zero.		
Assumptions:	• None		
Open issues:	• None		

Table 19: Remove FAQ Type - Technical Use Case Narrative

2.2.11 Create Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Stephan Brednehnn</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Role</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">2.11</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Administrator</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This administrative use case defines the creation of a new global catalogue role within the system. The Super Admin interacts with a specialized form interface to define the role's Name and Description. The Angular frontend transmits this payload to the secured “POST api/projectroles” endpoint. The backend's “IProjectRoleCatalogService” applies uniqueness validations before saving the new definition as a “ProjectRole” Entity using Entity Framework, thereby making the role available globally for future project assignments.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Administrator must be logged in with a valid JWT token.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin clicks the primary "Create Role" action button within the UI</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td>System Response</td><td></td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The Admin navigates to the Roles Management view and clicks the "Create Role" button.</td><td></td><td>Step 2: The Angular frontend triggers an event that opens a dynamic Dialog/Modal component layered over the active view. The system renders an Angular Reactive Form containing:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A "Role Name" standard text input field (configured with a mandatory required validator). • A "Role Description" multi-line text area input. • A secondary "Cancel" button. • A primary "Save Role" submit button.
	Step 3: The Admin populates the input fields, typing the desired Role Name and describing its intended responsibilities.		**Step 4:** The Reactive Form's internal state updates in real-time. Once the mandatory "Role Name" field is populated, the "Save Role" button's “disabled” state changes to false, allowing interaction.
	Step 5: The Admin confirms the details by clicking the "Save Role" button.		**Step 6:** The frontend extracts the form values and maps them into a JSON payload. The Angular service issues a secured “POST” request to the “api/projectroles” API endpoint, transmitting the “CreateProjectRoleDto” data structure in the request body.
			Step 7: The API's “ProjectRolesController” intercepts the request. Before

INTEGRATED NGO MANAGEMENT SYSTEM			
			executing business logic, the system executes the "access.EnsureCanManageCatalogAsync(User)" method to strictly verify the incoming JWT claims, ensuring the user possesses the correct authorization level to change catalogue roles.
			Step 8: The controller invokes the "IProjectRoleCatalogService.CreateAsync()" method. The backend system queries the database to enforce a critical business rule: it checks if any existing "ProjectRole" entity already uses the exact same "RoleName". [ALT]
			Step 9: Passing validation, the system instantiates a new ProjectRole entity in server memory. It maps the DTO properties to the entity framework model: **ProjectRole Entity:** • ProjectRoleId (Auto-generated PK) • ProjectId (Explicitly set to null to denote

INTEGRATED NGO MANAGEMENT SYSTEM			
			this as a global catalogue definition, rather than a scoped role) • RoleName (Mapped from payload) • RoleDescription (Mapped from payload) • IsActive (Iinitialised to true)
			Step 10: The system utilizes _dbContext.ProjectRoles.Add() to commit the new database record and returns a 'HTTP 201 Created' response.
			Step 12: The Angular frontend receives the successful response, dismisses the dialog modal, pushes the newly returned role object into the local state array bound to the data table (preventing a full page reload), and triggers an Angular Material Snack bar rendering a green success toast: "Role created successfully."
Alternate courses:	**[ALT] Step 8:** The backend database query detects an existing record with a matching RoleName. A response is returned with error message: "A catalogue role with this name already exists." The frontend prevents the modal from closing, highlights the Role Name input in red		
Conclusion:	A new role is added that can be used in the creation of any project on the system.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Post-condition:	A new, normalized ProjectRole entity is permanently committed to the SQL database with ProjectId assigned to null.
Business Rules:	• Project Role management.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 20: Create Role - Technical Use Case Narrative

2.2.12 View Role

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Stephan Brednehnn		**Date:** 28 April 2025 **Version:** 1.0
USE CASE NAME:	View Role		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	2.12		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case details the retrieval and visualization of the existing catalogue roles defined within the system. The Angular frontend initializes the Roles Management component, that dispatches a 'GET' request to the 'api/projectroles' endpoint. The backend's "IProjectRoleCatalogService" queries the database via Entity Framework Core, gathers all the "ProjectRole" entities, and transmits them back. The UI displays these roles within a data table.		
Pre-condition:	• The Administrator must be logged in with a valid JWT token.		
Trigger:	The Admin selects the "Roles Management" link from the system navigation bar.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Admin navigates into the Roles Management view.		**Step 2:** The Angular Router redirects to the Role Management module and sends a secured "GET" request to the "api/projectroles" API endpoint. **Step 3:** The "ProjectRolesController"

INTEGRATED NGO MANAGEMENT SYSTEM			
			receives the request. The system immediately executes "access.EnsureCanViewCatalogAsync(User)" to validate the presence of appropriate permissions embedded within the incoming JWT.
			Step 4: The controller invokes the "IProjectRoleCatalogService.ListAsync()" method. The backend runs an Entity Framework LINQ query targeting the "ProjectRole' table. The system filters the query specifically for records where the "ProjectId" property is "null" (filtering out localized project-scoped roles to guarantee only global catalogue definitions are fetched).
			Step 5: The system retrieves the matching ProjectRole entities from the database: **ProjectRole Entity Fields Retrieved:** • ProjectRoleId • RoleName • RoleDescription • IsActive
			Step 6: The system renders the UI, binding the data source to an Angular Material Data Table. The

INTEGRATED NGO MANAGEMENT SYSTEM			
			table constructs rows dynamically, displaying columns for: • Role Name • Role Description • Status (rendered as a dynamically coloured badge: Green for Active, Grey/Red for Inactive) • An "Actions" column populated with icon buttons tailored to the row's state (e.g., Edit button, Deactivate button if active, Reactivate button if inactive).
Alternate courses:	None		
Conclusion:	The administrator can view all global roles that can be assigned to projects.		
Post-condition:	The Angular front end table is fully populated with data.		
Business Rules:	• Project Role Management		
Implementation constraints and specifications:	• None		
Assumptions:	• There are Roles to populate the front end with.		
Open issues:	• None		

Table 21: View Role - Technical Use Case Narrative

2.2.13 Update Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Stephan Brednehnn</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Update Role</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">2.13</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Administrator</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the modification of pre-existing catalogue role definitions. The Admin initiates the flow from the data table, launching a pre-populated Angular Reactive Form. Edits are c routed to the “PUT api/projectroles/{id}” endpoint. The backend validates the payload, ensuring the uniqueness constraints are not violated, before applying changes directly to the target “ProjectRole” Entity and synchronizing the state with the database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Administrator must be logged in with a valid JWT token, and a role exists.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin clicks the "Edit" action icon button located on a specific row within the Roles Management data table.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The Admin locates the desired role in the data table and clicks the corresponding "Edit" icon button.</td><td></td><td>Step 2: The Angular frontend captures the selected role's ID. The system opens an "Edit Role" Modal component.</td></tr>
<tr><td></td><td></td><td></td><td>Step 3: The UI initializes a Reactive Form, pre-</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			populating the "Role Name" and "Role Description" input fields with the existing values passed from the table row.
	Step 4: The Admin modifies the values within the "Role Name" and/or "Role Description" fields, then clicks the "Update Role" button.		**Step 5:** The Reactive Form validates the input, ensuring mandatory fields remain populated. The frontend extracts the updated values, constructs the “UpdateProjectRoleDto” payload, and executes a secured “PUT” request to “api/projectroles/{id}”, where “{id}” corresponds to the “ProjectRoleId”.
			Step 6: The API's “ProjectRolesController” receives the request and checks access via “_access.EnsureCanManageCatalogAsync(User)”.
			Step 7: The controller hands over to “IProjectRoleCatalogService.UpdateAsync()”. The backend queries the database using “FindAsync” to fetch the specific “ProjectRole” entity

INTEGRATED NGO MANAGEMENT SYSTEM			
			corresponding to the provided ID.
			Step 8: The system executes a conflict check query. It scans the database to determine if the newly provided "RoleName" matches any other existing role (excluding the role currently being edited). [ALT]
			Step 9: Passing validations, the system overwrites the entity's properties in memory, updating "RoleName" and "RoleDescription" with the payload's strings.
			Step 10: The system executes "_dbContext.SaveChangesAsync()" to store the updated role details.
			Step 11: The API returns an "HTTP 200 OK" response containing the updated role object JSON.
			Step 12: The Angular frontend receives the successful HTTP response. The system closes the dialog modal. It locates the

INTEGRATED NGO MANAGEMENT SYSTEM			
			specific object within the local data table array and overwrites it with the returned object, triggering an immediate UI change of that specific row.
Alternate courses:	**[ALT] Step 8:** The uniqueness query discovers a conflict; the new "RoleName" is already claimed by a different entity. The API responds with an "HTTP 409 Conflict". The frontend keeps the modal open and displays an inline validation error prompting the user to choose a different name.		
Conclusion:	The role's name and/or description are successfully changed.		
Post-condition:	he specific "ProjectRole" entity retains its original Primary Key but contains an updated Name and/or description		
Business Rules:	• Project role management.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 22: Update Role - Technical Use Case Narrative

2.2.14 Reactivate Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Stephan Brednehnn</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">2.11</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">Reactivate Role</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Administrator</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case permits an Admin to restore a previously deactivated catalogue role to an active state, thereby enabling it to be assigned to users within projects again. The interaction is triggered via an action button in the data table. The Angular service issues a “PUT api/projectroles/{id}/reactivate” request. The backend locates the specific “ProjectRole” Entity and modifies its Boolean ‘IsActive’ attribute , concluding with an updated UI rendering the new state.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin must be authenticated. A target role must exist in the database and possess an “IsActive” property set to “false”.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin locates an inactive role and clicks the "Reactivate" button in the data table row.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The Admin clicks the "Reactivate" action button associated with an inactive role row.</td><td></td><td>Step 2: The Angular frontend intercepts the click event. To prevent accidental changes, the UI invokes a lightweight confirmation dialog ("Are you sure you wish</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			to reactivate this role?").
			Step 4: The frontend extracts the "ProjectRoleId" from the row's data context and sends "PUT" request directly to the "api/projectroles/{id}/reactivate" API endpoint.
			Step 5: The "ProjectRolesController" receives the incoming request and validates the administrator's JWT claims utilizing "_access.EnsureCanManageCatalogAsync(User)".
			Step 6: The controller invokes "IProjectRoleCatalogService.ReactivateAsync(id)". The backend issues a query to Entity Framework to locate the target "ProjectRole" entity using the provided ID.
			Step 8: The system invokes "_dbContext.SaveChangesAsync()" to commit the specific Boolean state change to the SQL database.

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 9: The API constructs an "HTTP 200 OK" response, embedding the updated JSON representation of the role within the body. **Step 10:** The Angular frontend processes the response. The data table's specific row object is updated. The UI changes the row, dynamically transitioning the status badge from a "Inactive" state to a "Active" state, and swaps the action button back to the "Deactivate" option.
Alternate courses:	None		
Conclusion:	The role is updated to show an active state and will once again appear for selection when creating a project.		
Post-condition:	The target 'ProjectRole' entity's 'IsActive' Attribute is changed to true		
Business Rules:	• Project Role Management		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 23: Reactivate Role - Technical Use Case Narrative

2.2.15 Deactivate Role

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Stephan Brednehnn		**Date:** 28 April 2025 **Version:** 1.0
USE CASE NAME:	Deactivate Role		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	2.12		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes the process of retiring a catalogue role. The system prevents destructive deletion to maintain historical integrity. Instead, the UI triggers a “PUT api/projectroles/{id}/deactivate” request. The backend’s “IProjectRoleCatalogService” performs mandatory relational dependency checks to guarantee the role is not actively assigned to users within ongoing projects. If clear, the “ProjectRole” Entity's status flag is toggled, hiding it from future selection when creating a project.		
Pre-condition:	• The Admin must be authenticated. The target role must be designated as active in the database.		
Trigger:	The Admin clicks the "Deactivate" icon button on a specific role within the Role Management data table.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Admin clicks the "Deactivate" button located on an active role's row.		**Step 2:** The Angular frontend detects the action. To prevent unintended changes, the UI displays a warning toast: "Deactivating this role

INTEGRATED NGO MANAGEMENT SYSTEM			
			will prevent it from being assigned in the future. Are you sure?"
	Step 3: The Admin reads the prompt and confirms the deactivation action. [ALT]		**Step 4:** The frontend extracts the "ProjectRoleId" and executes a "PUT" request targeting the "api/projectroles/{id}/ deactivate" endpoint.
			Step 5: The API receives the request. The controller asserts the administrator's JWT claims via "_access.EnsureCanManageCatalogAsync(User)"
			Step 6: "IProjectRoleCatalogService.DeactivateAsync(id)" is invoked. The backend queries the database using Entity Framework to locate the target "ProjectRole" entity.
			Step 7: The system executes a dependency analysis query. It actively scans tables such as "UserProjectRole" to verify if the role is currently bound to any users in active system projects. [ALT]
			Step 8: Upon confirming the role is

INTEGRATED NGO MANAGEMENT SYSTEM			
			safely unassigned and isolated, the system changes the entity in memory, setting the Boolean "IsActive" property to "false". **Step 9:** The system calls "dbContext.SaveChangesAsync()" to commit the deactivation transaction to the SQL database.
			Step 10: The API returns an "HTTP 200 OK" JSON response containing the modified role object.
			Step 11: The Angular frontend receives the successful payload. The UI updates the corresponding object within the data table. Angular's change detection triggers an update of changing the status badge from "Active" to "Inactive," and the button swaps to the "Reactivate" option.
Alternate courses:	**[ALT] Step 3:** The admin does not confirm, and the use case is terminated.		
	[ALT] Step 7: The rational dependency query detects that the role is actively assigned, the use case terminates with an error toast: "This role is still in use on an active project. Remove those team assignments or user project roles before deactivating."		

INTEGRATED NGO MANAGEMENT SYSTEM	
Conclusion:	The role is marked as deactivated and will no longer be available when creating a project.
Post-condition:	The “ProjectRole” Entity's “IsActive” attribute changes to false. It will no longer populate dropdown menus.
Business Rules:	• Role Management
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 24: Deactivate Role - Technical Use Case Narrative

2.2.16 Update Role Permissions

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 22 July 2026 **Version:** 1.0	
USE CASE NAME:	Update Role Permission	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	2.16		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	User		
Description:	This use case describes how an Administrator changes the permissions granted by a system-wide role (AppRole). The Administrator opens the role editor, toggles the permissions the role should hold, and saves. The Angular frontend sends the complete desired permission set to the ".NET 8" API endpoint ("PUT api/roles/{id}/permissions"), which replaces the role's entire permission set atomically whatever is sent is what the role gets. Built-in roles (User, Admin) can be re-permissioned freely, but the Super Admin role always rejects any restriction, since it must always hold every permission in the system.		
Pre-condition:	• The Administrator must be authenticated and hold the "Roles Manage" permission and must have navigated to the role editor for the role being changed.		
Trigger:	The Administrator toggles one or more permissions for a role and selects "Save Permissions".		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Administrator navigates to the roles management page.		**Step 2:** The system calls "GET api/roles" and displays each role with its name, description,

			user count, and permission count.
	Step 3: Administrator selects a role to edit.		**Step 4:** The system calls "GET api/roles/{id}" for the role's current permissions and "GET api/roles/permissions" for the full permission catalogue, then renders a permission grid grouped by module with the role's current permissions pre-toggled on. [ALT]
	Step 5: Administrator toggles permissions on or off for the role.		**Step 6:** The system tracks the toggle state and enables the "Save Permissions" button once it differs from the saved permission set.
	Step 7: Administrator selects "Save Permissions".		**Step 8:** The frontend RolesService sends "PUT api/roles/{id}/permissions" with only the permission keys currently toggled on.
			Step 9: The controller confirms the Administrator holds "Roles.Manage", then AppRoleService.SetRolePermissionsAsync loads the role together with its existing grants from: **RolePermission Entity:** • RolePermissionId (PK) • RoleId (FK)

			• PermissionKey
			Step 10: The service filters the incoming keys to only those recognised in the permission catalogue, normalises their casing, then diffs the current grants against the desired set, removing rows for keys no longer present and inserting rows for newly granted keys.
			Step 11: The API responds with "200 OK" and the updated role, including its new permission list. The frontend refreshes the permission grid and clears the unsaved-changes state.
Alternate courses:	**[ALT] Step 4:** The selected role is the Super Admin role. Every toggle is shown on, locked, and disabled, with the message "Super Admin always holds every permission and cannot be restricted." No save is possible for this role.		
	[ALT] Step 8: The Administrator's "Roles.Manage" permission was revoked after the page loaded. The API returns "403 Forbidden" and the frontend shows an inline alert.		
	[ALT] Step 9a: The role no longer exists. The API returns "404 Not Found" and the frontend shows an error.		
	[ALT] Step 9b: The role is the Super Admin role. The service rejects the change with "409 Conflict" ("Super Admin always holds every permission and cannot be restricted."), and the frontend displays the message inline.		
Conclusion:	The use case concludes when the role's permission set has been replaced and the updated grants are reflected in the permission grid.		

Post-condition:	The AppRolePermission rows for the role match exactly the submitted permission set. Users assigned to the role gain or lose the corresponding access on their next request, since permissions are resolved live from the database on every call.
Business Rules:	• The Super Admin role's permission set can never be reduced. • Only permission keys recognised in the catalogue are accepted; unrecognised keys are dropped silently. • Saving replaces the role's full permission set rather than merging it with the previous one.
Implementation constraints and specifications:	• "PUT api/roles/{id}/permissions" requires the "Roles.Manage" permission. • Permission keys are matched case-insensitively and normalised to the catalogue's casing on save.
Assumptions:	• Administrators understand that saving replaces the role's entire permission set rather than merging individual changes.
Open issues:	• None.

Table 25: Update Role Permission - Technical Use Case Narrative

2.2.17 View Role Permission

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 22 July 2026 **Version:** 1.0	
USE CASE NAME:	View Role Permission	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	2.17		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	User		
Description:	This use case describes how a user inspects what permissions each system-wide role grants. It covers the role list with permission summaries, drilling into a single role's full permission set, and browsing the complete permission catalogue grouped by module. It also covers the ".NET 8" API's "GET api/roles/me" call, which any authenticated user relies on to view their own effective permissions and gate their own interface.		
Pre-condition:	• The user must be authenticated. Viewing the role list, role detail, and permission catalog additionally requires the "Roles.View" permission; viewing one's own effective permissions requires only a valid session.		
Trigger:	The user navigates to the roles management page, opens a specific role, or logs in.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: User navigates to the roles management page.		**Step 2:** The system calls "GET api/roles" and displays every role as a card or table row, showing its name, a "Built-in" badge where applicable, description,

			user count, and permission count.
	Step 3: User selects a role to inspect.		**Step 4:** The system calls "GET api/roles/{id}" for the role's details and "GET api/roles/permissions" for the full catalog, then renders the role's permissions grouped by module, marking each one granted or not granted. [ALT]
	Step 5: User opens the standalone permission catalog view.		**Step 6:** The system calls "GET api/roles/permissions" and lists every permission the system recognises, grouped by module, without role-specific highlighting.
			Step 7: On login, the frontend calls "GET api/roles/me", which resolves the signed-in user's role and effective permissions from: **RolePermission Entity:** • RolePermissionId (PK) • RoleId (FK) • PermissionKey
			Step 8: The response is cached in the client-side auth store for the session and used to show or hide navigation items and

			action buttons the user is not permitted to use.
Alternate courses:	**[ALT] Step 2:** The user lacks "Roles.View". The API returns "403 Forbidden" and the "Roles" navigation item stays hidden.		
	[ALT] Step 4a: The selected role is the Super Admin role. Every permission is shown as granted, with the note "Super Admin implicitly holds every permission."		
	[ALT] Step 4b: The role no longer exists. The API returns "404 Not Found" and the page shows "This role does not exist. It may have been deleted."		
	[ALT] Step 7: The session's token is missing or expired. The API returns "401 Unauthorized" and the frontend redirects to the login page.		
Conclusion:	The use case concludes when the user has viewed the role list, a role's permission grants, the permission catalog, or their own effective permissions.		
Post-condition:	No data is changed; the user has viewed the requested role or permission information.		
Business Rules:	• "Roles.View" is required for the role list, role detail, and permission catalog endpoints. • "GET api/roles/me" requires only authentication. • The Super Admin role always returns every catalog key, regardless of its stored grants. • Permission keys are returned sorted alphabetically.		
Implementation constraints and specifications:	• "GET api/roles/permissions" is served from the in-memory permission catalog; it makes no database call. • A user's role change takes effect only after their next login, since the role is embedded in their JWT.		
Assumptions:	• Users understand that a role change made by an Administrator only takes effect for an affected user after that user's next login.		
Open issues:	• None.		

Table 26: View Role Permission - Technical Use Case Narrative

2.3 Inventory Subsystem Use Case Narratives

2.3.1 Create Procurement

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Procurement</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.1</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which an admin creates a procurement after receiving inventory. The admin will navigate to and select the Inventory tab and select the create procurement button. The system will load the Procurement screen where the admin will enter the procurements details. The admin will select create procurement and the system will validate the details entered before saving the procurement to the database. The use case ends once the system displays the success toaster and the procurement is saved on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Admin is logged in.</td></tr>
<tr><td>Trigger:</td><td colspan="3">Inventory is received and the admin requests to create a procurement.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Inventory is received and the admin requests to create a procurement.</td><td></td><td>Step 2: The system invokes use case 3.3 (View Inventory).</td></tr>
<tr><td>Step 3: The admin selects the “Create procurement” button.</td><td></td><td>Step 4: The Angular frontend calls the Warehouse, Vendor,</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			InventoryTypes and Projects service to send a GET request to the C# Web API backend Warehouse, Vendor, InventoryType and Projects controllers through the GetWarehouseLookup, GetVendorLookup, GetInventoryTypeLookup and GetProjectLookup endpoints. The backend retrieves active warehouse, vendor, inventory type and project details from the following entities: Warehouse Entity: • WarehouseId (PK) • Name Vendor Entity: • VendorId(PK) • Name Project Entity: • ProjectId (PK) • ProjectName InventoryType Entity: • InventoryTypeId (PK) • Name [ALT]
			Step 5: The system loads and displays the Procurement screen using the Angular frontend with

INTEGRATED NGO MANAGEMENT SYSTEM			
			the following controls: A header section containing: • A heading displaying Procurement. • A description displaying Receive inventory. A Procurement Details card containing: • A Vendor drop-down list populated with the available vendors retrieved from the backend. • A Warehouse drop-down list populated with the available warehouses retrieved from the backend. • A Procurement Date date picker for selecting the procurement date. • A Project (Optional) drop-down list populated with the available projects retrieved from the backend.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• An Invoice (PDF) file upload control for selecting a PDF invoice. • A label displaying the selected invoice filename or No file selected when no file has been selected. An Inventory Items section containing: • An Add Item button for adding a new inventory item row. • A table displaying one or more procurement item rows. Each procurement item row contains: • An Inventory Type drop-down list populated with the available inventory types retrieved from the backend. • A Model Name text box for entering the inventory model name. • A Quantity numeric input field for entering

INTEGRATED NGO MANAGEMENT SYSTEM			
			the quantity received. • A Price numeric input field for entering the unit price. • A Total field displaying the calculated line total (Quantity × Price). • A Remove button to remove the procurement item from the list. A summary section containing: • A Total display showing the calculated total value of all procurement items. An actions section containing: • A Cancel button to return to the Inventory screen. • A Create Procurement button to submit the procurement details and create the procurement record. An error message section containing:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A validation or processing error message displayed when an error occurs during procurement creation.
	Step 6: The admin enters the procurement's details and selects the "Create Procurement" button. [ALT]		Step 7: The system validates the details entered, validating the following: • All fields are completed. • Procurement date is not in the future. • At least one inventory item. • Inventory items are at least one quantity. • Inventory items have a model name. • Invoice is a PDF. [ALT]
			Step 8: The Angular frontend calls the Procurement service to send a POST request to the C# Web API backend Procurement controller through the CreateProcurement endpoint. The backend creates a new procurement in the following entities:

INTEGRATED NGO MANAGEMENT SYSTEM			
			Procurement Entity: • ProcurementId (PK) • VendorId (FK) • ProcurementDate • Invoice ProcurementLine Entity: • ProcurementLineId (PK) • ProcurementId (FK) • InventoryTypeId (FK) • ModelName • Quantity • Price Inventory • InventoryItemId (PK) • InventoryTypeId (FK) • ProcurementLineId (FK) • WarehouseId (FK) • ProjectId (FK) – if a project was selected. • ModelName • Value • Status [ALT]
			Step 9: The system will then record the transaction in the AuditLog table to maintain a history of actions

INTEGRATED NGO MANAGEMENT SYSTEM			
			performed within the system. This is handled in the backend after the successful creation of the procurement. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a procurement has been created. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 10: The system displays a success toaster informing the admin that the procurement was successfully created.
Alternate courses:	[ALT] Step 4: The system fails to retrieve the lookup data. The system will display an error message based on the status code.		
	[ALT] Step 6: The admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 7: The validation fails. The system displays an error message. Go back to step 6.		
	[ALT] Step 8: The system fails to create a new procurement. The system will display an error message based on the status code.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Conclusion:	The system displays a success toaster informing the admin that the procurement was successfully created.
Post-condition:	The procurement is saved on the system.
Business Rules:	• Super admins, admins or users with the required permissions can create procurements.
Implementation constraints and specifications:	• Each inventory item is stored individually in the Inventory table.
Assumptions:	• None
Open issues:	• None

Table 27: Create Procurement - Technical Use Case Narrativ

2.3.2 View Procurement

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Procurement</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.2</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which an admin views a procurement. The admin will navigate to the Inventory tab in the navbar. The admin will select a procurement, and the system will retrieve and display the procurement’s details on the system. The use case ends once the admin views the procurement.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Admin is logged into the system and procurement already exists.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The admin requests to view a procurement.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The admin requests to view a procurement.</td><td></td><td>Step 2: The system invokes use case 3.3 (View Inventory).</td></tr>
<tr><td>Step 3: The admin selects the procurement they wish to view.</td><td></td><td>Step 4: The Angular frontend calls the Procurement service to send a GET request to the C# Web API backend Procurement controller through the GetProcurementById</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			endpoint. The backend retrieves the procurement's details from the following entities: Procurement Entity: • ProcurementId (PK) • ProcurementDate • Invoice ProcurementLine Entity: • ProcurementLineId (PK) • ProcurementId (FK) • InventoryTypeId (FK) • ModelName • Quantity • Price InventoryType Entity: • InventoryTypeId (PK) • Name [ALT]
			Step 5: The system loads and displays the Procurement Details modal using the Angular frontend with the following controls populated with the selected procurement details:

INTEGRATED NGO MANAGEMENT SYSTEM			
			A header section containing: • A heading displaying the procurement number. A procurement details section containing: • A display of the procurement date. • A display of the total number of procurement items. • An Invoice section containing: • A View Invoice hyperlink for opening the uploaded invoice PDF in a new browser tab when an invoice exists. An Inventory Items section containing a table with the following columns: • Inventory Type displaying the inventory type name. • Model Name displaying the inventory model name. • Quantity displaying the quantity procured.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Price displaying the unit price. • Total displaying the calculated line total (Quantity × Price). A summary row containing: • A Grand Total display showing the calculated total value of all procurement items. A footer section containing: • A Close button to close the Procurement Details modal and return to the Inventory screen.
Alternate courses:	[ALT] Step 4: The system fails to retrieve the procurement's details. The system will display an error toaster message based on the status code.		
Conclusion:	The system displays the procurement's details.		
Post-condition:	The procurement's details is displayed.		
Business Rules:	• Super admins, admins or users with the required permissions can view procurements.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		

INTEGRATED NGO MANAGEMENT SYSTEM	
Open issues:	• None

Table 28:Table 27: View Procurement - Technical Use Case Narrative

2.3.3 View Inventory

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Christopher Bailey		**Date: 13 July 2026** **Version: 1**
USE CASE NAME:	View Inventory		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	3.3		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	• None		
Description:	This use case describes the event in which an admin views inventory. The admin will navigate to the Inventory tab in the navbar and the system will retrieve and display all inventory on the system where the admin can filter and search inventory. The use case ends once the admin views inventory.		
Pre-condition:	Admin is logged into the system.		
Trigger:	The admin requests to view inventory.		
Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: The admin requests to view inventory.		
	Step 2: The admin navigates to and selects the Inventory tab. [ALT]		Step 3: The Angular frontend calls the Warehouse and InventoryTypes service to send a GET request to the C# Web API backend Warehouse and InventoryType controllers

INTEGRATED NGO MANAGEMENT SYSTEM			
			through the GetWarehouseLookup, GetInventoryTypeLookup endpoints. The backend retrieves active warehouse, inventory type details from the following entities: Warehouse Entity: • WarehouseId (PK) • Name InventoryType • InventoryTypeId (PK) • Name [ALT]
			Step 4: The Angular frontend calls the Inventory service to send a GET request to the C# Web API backend Inventory controller through the GetQuantityTotals endpoint. The backend retrieves the inventory from the following entities: Inventory Entity: • InventoryItemId (PK) • InventoryTypeId (FK) • ModelName • Value • Status InventoryType Entity:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• InventoryTypeId (PK) • Name The backend groups the inventory records by InventoryTypeId, Inventory Type Name, and Model Name, and calculates the quantity and total value for each inventory status. [ALT]
			Step 5: The Angular frontend calls the Inventory service to send a GET request to the C# Web API backend Inventory controller through the GetAll endpoint. The backend retrieves the inventory from the following entities: Inventory Entity • InventoryItemId (PK) • InventoryTypeId (FK) • ProcurementLineId (FK) • WarehouseId (FK) • ProjectId (FK) • ModelName • Value • Status InventoryType Entity: • InventoryTypeId (PK)

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Name Warehouse Entity: • WarehouseId (PK) • Name • Location ProcurementLine Entity: • ProcurementLineId (PK) • ProcurementId (FK) [ALT]
			Step 6: The system displays the Quantity Totals screen using the Angular frontend with the following controls: A Quantity Totals Filters section containing: • An Inventory Types multi-select control populated with the available inventory types. • A Status drop-down list containing the available inventory statuses. • A View Inventory button to return to the Inventory view. A Quantity Totals table containing the following columns: • Inventory Type displaying the

INTEGRATED NGO MANAGEMENT SYSTEM			
			inventory type name. • Model Name displaying the inventory model name. • Total Quantity displaying the total quantity of inventory items. • Total Value displaying the total monetary value of the inventory items. An Actions column containing: • An Allocate button for allocating available inventory items to a project. • A Write Off button for writing off available inventory items. The Actions column is displayed only when the selected inventory status is Available. A fallback message displaying "No inventory found." when no inventory records match the selected filters.
	Step 7: The admin selects the "View Inventory" button. [ALT]		Step 8: The system loads and displays the Inventory screen using the Angular

INTEGRATED NGO MANAGEMENT SYSTEM			
			frontend with the following controls: A header section containing: • A heading displaying Inventory. • A description displaying View and manage inventory. • A Manage Inventory Types button to navigate to the Inventory Type Management screen. • A Create Procurement button to navigate to the Procurement screen. An Inventory Filters section containing: • A Search Inventory search box for filtering inventory records. • A Status drop-down list containing the available inventory statuses. • An Inventory Type drop-down list populated with the available inventory types.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Warehouse drop-down list populated with the available warehouses. • A View Quantity Totals button to switch to the Quantity Totals view. An Inventory table containing the following columns: • Inventory Type displaying the inventory type name. • Model Name displaying the inventory model name. • Warehouse displaying the warehouse name. • Location displaying the warehouse location. • Procurement ID displayed as a button for viewing the associated procurement details. • Value displaying the monetary value of the inventory item.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Status displaying the current inventory status. A fallback message displaying "No inventory found." when no inventory records are available.
	Step 9: The admin filters the inventory. [ALT]		Step 10: The frontend Angular filters the inventory based on the applied criteria. [ALT]
Alternate courses:	[ALT] Step 2: The admin is using a mobile device such as a phone. The admin will select the menu (hamburger) icon and then the inventory option. Continue to step 3.		
	[ALT] Step 3: The system fails to retrieve the lookup data. The system will display an error toaster message based on the status code.		
	[ALT] Step 4: The system fails to retrieve the inventory quantity totals. The system will display an error toaster message based on the status code.		
	[ALT] Step 5: The system fails to retrieve inventory. The system will display an error toaster message based on the status code.		
	[ALT] Step 7a: The admin does not select the View Inventory button. The system remains on the Quantity Totals view. [ALT] Step 7b: The admin filters the inventory. The frontend Angular filters the inventory using the applied filters and displays the inventory matching the criteria.		
	[ALT] Step 9: The admin does not apply any filters.		
	[ALT] Step 10: No inventory matches the applied filters. Go back to step 9.		
Conclusion:	The system displays the inventory.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Post-condition:	Inventory is displayed.
Business Rules:	• Super admins, admins or users with the required permissions can view inventory.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 29: View Inventory - Technical Use Case Narrative

2.3.4 Write-Off Inventory

INTEGRATED NGO MANAGEMENT SYSTEM		
AUTHOR:	Christopher Bailey	**Date: 13 July 2026** **Version: 1**
USE CASE NAME:	Write-Off Inventory	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	3.4	
PRIORITY:	Medium	
SOURCE:	EPI-USE	
Primary Business Actor (PBA):	Admin	
Primary System Actor (PSA):	• None	
Other Participating Actors:	• None	
Other Interested Stakeholders:	• None	
Description:	This use case describes the event in which an admin writes-off inventory. The admin will navigate to the Inventory tab in the navbar. The system displays the inventory records and allows the admin to select inventory for the write-off process. The system then displays the Write Off Inventory modal, where the admin enters the quantity of inventory to be written off. The system validates the entered quantity before updating the status of the selected inventory items to Written Off. The use case	

INTEGRATED NGO MANAGEMENT SYSTEM			
	ends when a success toast notification is displayed, confirming that the selected inventory items have been successfully updated.		
Pre-condition:	Admin is logged into the system.		
Trigger:	The admin requests to write-off inventory.		
Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: The admin requests to write-off inventory.		Step 2: The system invokes use case 3.3 (View Inventory).
	Step 3: The admin selects the "Write Off" button.		Step 4: The system displays the Write Off modal using the Angular frontend with the following controls populated with the selected inventory details: A header section containing: • A heading displaying Write Off Inventory. A form section containing: • An Inventory Type read-only text box displaying the selected inventory type. • A Model Name read-only text box displaying the selected model name. • An Available Quantity read-only text box displaying

INTEGRATED NGO MANAGEMENT SYSTEM			
			the quantity of available inventory items. • A Quantity to Write Off numeric input field for entering the quantity of inventory items to be written off. • A Reason for Write Off input field. An error message section containing: • A validation or processing error message displayed when an error occurs during the write-off process. A footer section containing: • A Cancel button to close the Write Off Inventory modal without making any changes. • A Write Off button to submit the write-off request.
	Step 5: The admin enters the amount of inventory to write-off and the reason for write off, and selects the "Write Off" button. [ALT]		Step 6: The system validates the details entered, validating the following: • The quantity entered does not

INTEGRATED NGO MANAGEMENT SYSTEM			
			exceed the quantity available. • The write off reason field is completed. [ALT]
			Step 7: The Angular frontend calls the Inventory service to send a POST request to the C# Web API backend Inventory controller through the WriteOffInventory endpoint. The backend updates the matching inventory item(s) in the following entity: Inventory Entity: • Status The backend retrieves the inventory items matching the specified InventoryTypeId, ModelName, and Available status, up to the requested quantity and updates their status to Written Off. [ALT]
			Step 8: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the

INTEGRATED NGO MANAGEMENT SYSTEM			
			successful update of the inventory item(s) status. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that inventory has been written off. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a success toaster informing the admin that the inventory was successfully written off.
Alternate courses:	[ALT] Step 5: The admin selects the "Cancel" button. Use case terminates.		
	[ALT] Step 6: The validation fails. The system displays an error message. Go back to step 5.		
	[ALT] Step 7: The system fails to update the inventory item(s) status. The system will display an error message based on the status code.		
Conclusion:	A success toaster is displayed informing the admin that inventory was successfully written off.		
Post-condition:	Inventory is successfully written off.		
Business Rules:	• Super admins, admins or users with the required permissions can write off inventory. • Only available inventory can be written off.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 30: Write-Off Inventory - Technical Use Case Narrative

2.3.5 Allocate Inventory

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Allocate Inventory</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.5</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which an admin allocates inventory. The admin will navigate to the Inventory tab in the navbar. The system displays the inventory records and allows the admin to select inventory for the allocation process. The system then displays the Allocate Inventory modal, where the admin selects an active project and enters the quantity of inventory to be allocated. The system validates the details before updating the status of the selected inventory items to Allocated. The use case ends when a success toast notification is displayed, confirming that the selected inventory items have been successfully allocated.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The admin requests to allocate inventory.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The admin requests to allocate inventory.</td><td></td><td>Step 2: The system invokes use case 3.3 (View Inventory).</td></tr>
<tr><td>Step 3: The admin selects the “Allocate” button.</td><td></td><td>Step 4: The Angular frontend calls the Project service to send a GET request to the C# Web API</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			backend Project controller through the GetProjectLookup endpoint. The backend retrieves active projects from the following entity: Project Entity: • ProjectId (PK) • ProjectName [ALT]
			Step 5: The system displays the Allocate modal using the Angular frontend with the following controls populated with the selected inventory details: A header section containing: • A heading displaying Allocate Inventory. A form section containing: • An Inventory Type read-only text box displaying the selected inventory type. • A Model Name read-only text box displaying the selected model name. • An Available Quantity read-only text box displaying the quantity of

INTEGRATED NGO MANAGEMENT SYSTEM			
			available inventory items. • A Project drop-down list populated with the available projects. • A Quantity numeric input field for entering the quantity of inventory items to allocate. An error message section containing: • A validation or processing error message displayed when an error occurs during the allocation process. A footer section containing: • A Cancel button to close the Allocate Inventory modal without making any changes. • An Allocate button to submit the inventory allocation request.
	Step 6: The admin enters the amount of inventory to allocate as well as select a project then selects the "Allocate" button. [ALT]		Step 7: The system validates the details entered, validating the following: • A project is selected.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• The quantity entered does not exceed the quantity available. [ALT]
			Step 8: The Angular frontend calls the Inventory service to send a POST request to the C# Web API backend Inventory controller through the AllocateInventory endpoint. The backend updates the matching inventory item(s) in the following entity: Inventory Entity: • Status • ProjectId (FK) The backend retrieves the inventory items matching the specified InventoryTypeId, ModelName, and Available status, up to the requested quantity and updates their status to Allocated. [ALT]
			Step 9: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the inventory item(s) status.

INTEGRATED NGO MANAGEMENT SYSTEM			
			The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that inventory has been allocated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 10: The system displays a success toaster informing the admin that the inventory was successfully written off.
Alternate courses:	[ALT] Step 4: The system fails to retrieve the lookup data. The system will display an error message based on the status code.		
	[ALT] Step 6: The admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 7: The validation fails. The system displays an error message. Go back to step 6.		
	[ALT] Step 8: The system fails to update the inventory item(s) status. The system will display an error message based on the status code.		
Conclusion:	The inventory item(s) is successfully allocated.		
Post-condition:	Inventory is successfully allocated to a project.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Business Rules:	• Super admins, admins or users with the required permissions can allocate inventory. • Only available inventory can be allocated. • Inventory can only be allocated to active projects.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 31: Allocate Inventory - Technical Use Case Narrative

2.3.6 Create Inventory Item Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Inventory Item Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.6</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin creates an inventory item type. The system will display inventory type records and allow the super admin to create a new type. The system will display the Create Inventory Item Type modal and allow the super admin to enter the new inventory type’s details. The use case ends once the success toaster is displayed and the new inventory item type has successfully been captured on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to create a new inventory item type.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to create a new inventory item type.</td><td></td><td>Step 2: The system invokes use case 3.7 (View Inventory Item Types).</td></tr>
<tr><td>Step 3: The super admin selects “Create” button.</td><td></td><td>Step 4: The system loads and displays the Create Inventory Item Type modal using the Angular frontend with the following controls:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			A header section containing: • A heading displaying Create Inventory Type. A form section containing: • A Name text box for entering the inventory type name. • A Security Level drop-down list containing the available security levels (Low, Medium, and High). • A Description text area for entering the inventory type description. • A checkbox to indicate whether the type is a depreciating asset. An error message section containing: • A validation or processing error message displayed when an error occurs during the inventory type creation process. A footer section containing:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Cancel button to close the Create Inventory Type modal without creating a new inventory type. • A Create button to submit the inventory type details. The button remains disabled until all required fields have been completed.
	Step 5: The super admin enters the new inventory item type's details and selects the "Create" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All fields are completed [ALT]
			Step 7: The Angular frontend calls the InventoryTypes service to send a POST request to the C# Web API backend InventoryType controller through the CreateInventoryType endpoint. The backend creates the inventory item type in the following entity: InventoryType Entity: • Name • Description • SecurityLevel

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Status [ALT]
			Step 8: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the inventory item type. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that an inventory item type was created. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a success toaster informing the admin that the inventory item type was successfully created.
Alternate courses:	[ALT] Step 5: The super admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 6:		

INTEGRATED NGO MANAGEMENT SYSTEM	
	The validation fails. The system displays an error message. Go back to step 5.
	[ALT] Step 7: The system fails to create the inventory item type. The system will display an error message based on the status code.
Conclusion:	A success toaster is displayed informing the super admin that the inventory item type was successfully created.
Post-condition:	The inventory item type is saved on the system.
Business Rules:	• Super admins or users with the required permissions can create inventory item types.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 32: Create Inventory Item Type - Technical Use Case Narrative

2.3.7 View Inventory Item Types

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Inventory Item Types</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.7</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin views inventory item types. The super admin will navigate to the Inventory tab in the navbar. The system displays the inventory records and allows the super admin to select the Manage inventory types button. The system will retrieve and display inventory type records. The use case ends once the inventory item types are displayed.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to view inventory item types.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to view inventory item types.</td><td></td><td>Step 2: The system invokes use case 3.3 (View Inventory).</td></tr>
<tr><td>Step 3: The admin selects the “Manage inventory types” button.</td><td></td><td>Step 4: The Angular frontend calls the InventoryTypes service to send a GET request to the C# Web API backend InventoryType controller through the</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			GetAllInventoryTypes endpoint. The backend retrieves the inventory item types from the following entity: InventoryType Entity: • InventoryTypeId (PK) • Name • Description • SecurityLevel • Status [ALT]
			Step 5: The system loads and displays the Inventory Type screen using the Angular frontend with the following controls: A header section containing: • A heading displaying Inventory Types. • A description displaying Create, edit, and manage inventory categories. • A Back button to return to the Inventory screen. • A Create button to open the Create Inventory Type modal.

INTEGRATED NGO MANAGEMENT SYSTEM			
			An Inventory Types table containing the following columns: • Name displaying the inventory type name. • Description displaying the inventory type description. • Security Level displaying the inventory type security level. • Status displaying the current inventory type status (Active or Inactive). An Actions column containing: • An Update button to open the Update Inventory Type modal. The button is disabled when the inventory type is inactive. • A Deactivate button to deactivate an active inventory type. • A Reactivate button to reactivate an

INTEGRATED NGO MANAGEMENT SYSTEM			
			inactive inventory type.
Alternate courses:	[ALT] Step 4: The system fails to retrieve the inventory item types. The system will display an error toaster message based on the status code.		
Conclusion:	The inventory item types are displayed.		
Post-condition:	The inventory item types are successfully displayed.		
Business Rules:	• Super admins or users with the required permissions can view inventory item types.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 33: View Inventory Item Types - Technical Use Case Narrative

2.3.8 Update Inventory Item Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Update Inventory Item Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.8</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin updates an inventory item type. The super admin will navigate to and select the Manage inventory types button and the system will retrieve and display inventory type records. The super admin will select the inventory item type they wish to update, and the system will display the Update Inventory Item Type modal. The super admin will then enter the updated details. The use case ends once the success toaster is displayed and the inventory item type is updated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.
The inventory item type exists.
The inventory item type is active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to update an inventory item type.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to update an inventory item type.</td><td></td><td>Step 2: The system invokes use case 3.7 (View Inventory Item Types).</td></tr>
<tr><td>Step 3: The admin selects the “Update” button on the inventory item type they wish to update.</td><td></td><td>Step 4: The Angular frontend calls the InventoryTypes service to send a GET request to the</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			C# Web API backend InventoryType controller through the GetInventoryTypeById endpoint. The backend retrieves the inventory item type from the following entity: InventoryType Entity: • InventoryTypeId (PK) • Name • Description • SecurityLevel • Status [ALT]
			Step 5: The system displays the Update Inventory Item Type modal using the Angular frontend with the following controls populated with the selected inventory type details: A header section containing: • A heading displaying Update Inventory Type. A form section containing: • A Name text box displaying the inventory type name. • A Security Level drop-down list displaying the

INTEGRATED NGO MANAGEMENT SYSTEM			
			current security level and allowing the super admin to select Low, Medium, or High. • A Description text area displaying the inventory type description. An error message section containing: • A validation or processing error message displayed when an error occurs during the inventory type update process. A footer section containing: • A Cancel button to close the Update Inventory Type modal without saving any changes. • A Save Changes button to submit the updated inventory type details. The button remains disabled until all required fields have been completed.
	Step 6: The super admin enters the updated details		Step 7: The system validates the details

INTEGRATED NGO MANAGEMENT SYSTEM			
	and selects the "Save Changes" button. [ALT]		entered, validating the following: • All fields are completed [ALT]
			Step 8: The Angular frontend calls the InventoryTypes service to send a PUT request to the C# Web API backend InventoryType controller through the UpdateInventoryType endpoint. The backend updates the inventory item type in the following entity: InventoryType Entity: • Name • Description • SecurityLevel [ALT]
			Step 9: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the inventory item type. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a

INTEGRATED NGO MANAGEMENT SYSTEM			
			descriptive log entry indicating that an inventory item type was updated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 10: The system displays a success toaster informing the admin that the inventory item type was successfully updated.
Alternate courses:	[ALT] Step 4: The system fails to retrieve the inventory item type. The system will display an error toaster message based on the status code.		
	[ALT] Step 6: The super admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 7: The validation fails. The system displays an error message. Go back to step 6.		
	[ALT] Step 8: The system fails to update the inventory item type. The system will display an error message based on the status code.		
Conclusion:	A success toaster is displayed to inform the super admin that the update was successful.		
Post-condition:	The inventory item type is successfully updated.		
Business Rules:	• Super admins or users with the required permissions can update inventory item types.		
Implementation constraints and specifications:	• None		

INTEGRATED NGO MANAGEMENT SYSTEM	
Assumptions:	• None
Open issues:	• None

Table 34: Update Inventory Item Type - Technical Use Case Narrative

2.3.9 Deactivate Inventory Item Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Inventory Item Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.9</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin deactivates an inventory item type. The super admin will navigate to and select the Manage inventory types button and the system will retrieve and display inventory type records. The super admin will select the inventory item type they wish to deactivate, and the system will display the Confirm Inventory Item Type Deactivation modal. The super admin will then confirm the deactivation. The use case ends once the success toaster is displayed and the inventory item type is deactivated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.
The inventory item type is active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to deactivate an inventory item type.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to deactivate an inventory item type.</td><td></td><td>Step 2: The system invokes use case 3.7 (View Inventory Item Types).</td></tr>
<tr><td>Step 3: The admin selects the “Deactivate” button on the inventory item type they wish to deactivate.</td><td></td><td>Step 4: The system displays the Confirm Inventory Item Type Deactivation modal using</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			the Angular frontend with the following controls: A header section containing: • A heading displaying Deactivate Inventory Type. A confirmation section containing: • A confirmation message asking the administrator to confirm the deactivation of the selected inventory type. • A warning message indicating that the inventory type will no longer be available for use when creating new inventory items. An error message section containing: • A validation or processing error message displayed when an error occurs during the deactivation process. A footer section containing:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Cancel button to close the confirmation modal without deactivating the inventory type. • A Confirm button to deactivate the selected inventory type.
	Step 5: The super admin selects the "Confirm" button. [ALT]		Step 6: The Angular frontend calls the InventoryTypes service to send a PUT request to the C# Web API backend InventoryType controller through the DeactivateInventoryType endpoint. The backend updates the selected inventory item type's status in the following entity: InventoryType Entity: • Status [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the inventory item type. The system will store information such as the

INTEGRATED NGO MANAGEMENT SYSTEM			
			authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that an inventory item type was deactivated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a success toaster informing the admin that the inventory item type was successfully deactivated.
Alternate courses:	[ALT] Step 5: The super admin selects the "Cancel" button. Use case terminates.		
	[ALT] Step 6: The system fails to deactivate the inventory item type. The system will display an error message based on the status code.		
Conclusion:	A success toaster is displayed informing the super admin that the inventory item type was successfully deactivated.		
Post-condition:	• The inventory item type is successfully deactivated. • New procurements will no longer be able to use the deactivated inventory item type.		
Business Rules:	• Super admins or users with the required permissions can deactivate inventory item types.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		

INTEGRATED NGO MANAGEMENT SYSTEM	
Open issues:	• None

Table 35: Deactivate Inventory Item Type - Technical Use Case Narrative

2.3.10 Reactivate Inventory Item Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 13 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Reactivate Inventory Item Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">3.10</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin reactivates an inventory item type. The super admin will navigate to and select the Manage inventory types button and the system will retrieve and display inventory type records. The super admin will select the inventory item type they wish to reactivate, and the system will display the Confirm Inventory Item Type Reactivation modal. The super admin will then confirm the reactivation. The use case ends once the success toaster is displayed and the inventory item type is reactivated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.
The inventory item type is inactive.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to reactivate an inventory item type.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to reactivate an inventory item type.</td><td></td><td>Step 2: The system invokes use case 3.7 (View Inventory Item Types).</td></tr>
<tr><td>Step 3: The admin selects the “Reactivate” button on the inventory item type they wish to reactivate.</td><td></td><td>Step 4: The system displays the Confirm Inventory Item Type Reactivation modal using</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			the Angular frontend with the following controls: A header section containing: • A heading displaying Reactivate Inventory Type. A confirmation section containing: • A confirmation message asking the administrator to confirm the reactivation of the selected inventory type. • An information message indicating that the inventory type will be available for use when creating new inventory items. An error message section containing: • A validation or processing error message displayed when an error occurs during the reactivation process. A footer section containing: • A Cancel button to close the

INTEGRATED NGO MANAGEMENT SYSTEM			
			confirmation modal without reactivating the inventory type. • A Confirm button to reactivate the selected inventory type.
	Step 5: The super admin selects the "Confirm" button. [ALT]		Step 6: The Angular frontend calls the InventoryTypes service to send a PUT request to the C# Web API backend InventoryType controller through the ReactivateInventoryType endpoint. The backend updates the selected inventory item type's status in the following entity: InventoryType Entity: • Status [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful reactivation of the inventory item type. The system will store information such as the authenticated user performing the action, the

INTEGRATED NGO MANAGEMENT SYSTEM			
			timestamp of the operation, and a descriptive log entry indicating that an inventory item type was reactivated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a success toaster informing the admin that the inventory item type was successfully reactivated.
Alternate courses:	[ALT] Step 5: The super admin selects the "Cancel" button. Use case terminates.		
	[ALT] Step 6: The system fails to reactivate the inventory item type. The system will display an error message based on the status code.		
Conclusion:	The inventory item type is reactivated.		
Post-condition:	• The inventory item type is successfully reactivated. • New procurements will be able to use the reactivated inventory item type.		
Business Rules:	• Super admins or users with the required permissions can reactivate inventory item types.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 36: Receive Inventory Item Type - Technical Use Case Narrative

2.4 Group Subsystem Use Case Narratives

2.4.1 Create Group

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Group</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.1</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• Users</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin creates a new group within a selected project. The system allows the user to enter the group’s details, assign users, and then the system will validate the input, and confirm the creation before saving the group to the database. The group is then stored, linked to each user, and a confirmation message is displayed. The transaction is also recorded in the audit log.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to add a new group to the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to add a new group to the system.</td><td></td><td>Step 2: The system invokes UC 4.2 (View Groups) to display a pop-up showing all the groups’ details. This is done using Angular frontend and the group service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			endpoint in Visual Studio. (Group Controller)
	Step 3: The super admin selects the "Add Group" button.		**Step 4:** The system will then load and display the Create Group Modal with the following components: • Create Group modal title • Group name section: ○ Group name label and input field ○ Placeholder text: "Enter group name" • User selection section: ○ Add user label ○ Dropdown list ▪ Default option: "Select user" ▪ Displays username and user surname • Add button (Disabled when no user is selected) • Selected users section: ○ List of selected users displayed as tags ○ Displays username and surname

INTEGRATED NGO MANAGEMENT SYSTEM			
			○ Remove (×) Button for each user • Action buttons section: • Create button (Disabled if the group name is empty) • Cancel button The User dropdown list is populated using data retrieved from the User Entity via the User Service, which calls the backend API. **User Entity:** • UserId (PK) • UserName • UserSurname
	Step 5: The super admin enters the group's details, adds users and then selects the "Create" button. [ALT]		**Step 6:** The system validates the entered details using Angular form validation and frontend checks. The system validates the following: • The Group Name input field is not null or empty. • The Group Name input does not contain only whitespace characters. • Duplicate users are not added to the selectedUsers list. [ALT]
			Step 7: The system displays a custom Angular confirmation modal, which is triggered when

INTEGRATED NGO MANAGEMENT SYSTEM			
			the Super Admin clicks the "Create" button. • Confirmation modal title ("Confirm group creation") • Confirmation message label ○ Displays message: "Are you sure you want to create this group?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system will then capture the new Group's details and route the data to the corresponding API endpoints using the Angular frontend and group service to route the request to the appropriate HTTP POST endpoint in Visual Studio. (GroupController – CreateGroup). The API endpoint will validate the incoming data and create the new Group in the database using Entity Framework Core in MS SQL Server. The following entities are involved in the process: **Group Entity:** • GroupId (PK)

INTEGRATED NGO MANAGEMENT SYSTEM			
			• ProjectId (FK) • GroupName • ActiveStatus • DateCreated **UserGroup Entity (Junction Table):** • UserId (PK, FK) • GroupId (PK, FK) [ALT]
			Step 10: The system will synchronise the newly created Group with Firestore to support real time messaging between users.
			Step 11: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the new group. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a new group has been created. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp

INTEGRATED NGO MANAGEMENT SYSTEM			
			• LogEntry
			Step 12: The system will then display a confirmation message indicating that the group has been successfully created. This is implemented in the Angular frontend using a toaster notification component. The confirmation message will display the text: "Group created successfully".
Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and displays an error message within the Create Group modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 4.		
	ALT-Step 9: The system fails to save the new group and displays a toaster message: "Failed to save new group". Terminate use case.		
Conclusion:	The new group's details have been saved onto the system, and a confirmation message is shown.		
Post-condition:	The system saves the new group onto the Group table and the assigned users onto the UserGroup table.		
Business Rules:	• Only super admin or users given appropriate permissions can add a new group.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 37: Create Group - Technical Use Case Narrative

2.4.2 View Groups

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Groups</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.2</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes when a user wants to view the groups for a specific project on the system The system retrieves active projects along with their group count and displays them in a card using the Angular frontend. The user can select a project, and the system retrieves all related groups, including their details and assigned users through the relevant API endpoints and displayed in a pop-up modal. The groups are shown with their names and members.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user wants to view the groups for a specific project to which they are assigned.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user wants to view the groups for a specific project to which they are assigned.</td><td></td><td></td></tr>
<tr><td>Step 2: The user selects the “Teams” tab to navigate to</td><td></td><td>Step 3: The system sends the request through the Angular frontend and project card</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	the teams' page. [ALT]		service to the relevant HTTP GET API endpoint in Visual Studio (GroupController – GetProjectCards) which retrieves all projects and their associated group counts from the Project and Group tables in the database. **Project Entity:** • ProjectId (PK) • ProjectName **Group Entity:** • GroupId (PK) • ProjectId (FK) • GroupName • ActiveStatus • DateCreated Only active groups with are included in the results.
			Step 4: The system displays a list of projects and the number of groups associated with each project based on the user's role permissions. The retrieved projects and their number of groups are shown in the Angular frontend using a dynamic card layout. The system will load and display the cards with the following components: • Page title:

INTEGRATED NGO MANAGEMENT SYSTEM			
			○ "Active Projects" • Project card for each project: ○ Project name ○ Group count ○ Clickable card event (Allows the user to select a project) The system ensures that only projects and groups that the user is assigned to are displayed. [ALT]
	Step 5: The user selects a project card.		**Step 6:** The system retrieves all groups associated with the selected project, including their details and the users assigned to each group through the group service in Angular frontend. The service sends a request via Angular HTTP Client to the HTTP GET endpoint in Visual Studio (GroupController – GetByProject). The API then retrieves data from the following tables: **Group Entity:** • GroupId (PK) • ProjectId (FK) • GroupName • ActiveStatus • DateCreated **UserGroup Entity (Junction Table):**

INTEGRATED NGO MANAGEMENT SYSTEM			
			• UserId (PK, FK) • GroupId (PK, FK) **User Entity:** • UserId (PK) • UserName • UserSurname
			Step 7: The system displays a pop-up modal on the Angular frontend containing all the retrieved group details for the selected project using an Angular modal that is triggered when a project card is selected. The view modal has the following components: • Modal heading: ○ Project name The system displays each group with the following details: • Group name • Group actions: ○ Edit button ○ Deactivate button • Group members section: ○ Displays assigned users ○ Each user is shown as a tag containing:

INTEGRATED NGO MANAGEMENT SYSTEM			
			▪ Username ▪ User Surname The modal shows a view of all active groups within the selected project. [ALT]
	Step 8: The user can view the groups.		
Alternate courses:	**ALT-Step 2:** User is on mobile and selects the menu icon. Go to step 3.		
	ALT-Step 4: The system does not find any active projects. Terminate use case.		
	ALT-Step 7: The system does not find any active groups in the selected project. Terminate use case.		
Conclusion:	The system displays all the groups' details.		
Post-condition:	All the related groups' details are displayed.		
Business Rules:	• Users can only view groups they are assigned to. • Admins and super admins can view all groups on the system.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 38: View Groups - Technical Use Case Narrative

2.4.3 Update Group

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Update Group</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.3</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin updates an existing group within a selected project. The system retrieves and displays the current group details, including the group name and assigned users. The Super Admin can update the group name and manage group membership by adding or removing users. Once the changes are made, the system validates the updated information before submitting it to the backend API. The use case ends when the system successfully saves the updated group’s details to the database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to update a group’s details on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to update a group’s details on the system.</td><td></td><td>Step 2: The system invokes UC 4.2 (View Groups) to display a pop-up showing all the groups’ details. This is done using Angular frontend and the group service, which routes the request via Angular HTTP Client to the corresponding</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			HTTP GET API endpoint in Visual Studio. (Group Controller)
	Step 3: The super admin selects the "Edit" button on a specific group.		**Step 4:** The system displays the Edit Group Modal with the group name pre-populated and editable. The modal contains the following components: • Edit Group modal title • Group name section: ○ Input field ○ Pre-populated with the selected group's current name ○ Allows the user to modify the group name • Add User Section: ○ Dropdown list populated via the user service ○ Displays: ▪ Username ▪ User surname ○ Add Button

INTEGRATED NGO MANAGEMENT SYSTEM			
			○ Prevents duplicate selection • Users Section: ○ Displays a list of all users currently assigned to the group ○ Each user is shown as a tag containing: ▪ Username ▪ User surname ○ Each user tag includes a remove button The User dropdown list is populated using data retrieved from the User Entity via the User Service, which calls the backend API. **User Entity:** • UserId (PK) • UserName • UserSurname
	Step 5: The super admin updates the group's details and selects the "Save" button. [ALT]		**Step 6:** The system validates the updated group's details using Angular form validation and frontend checks before allowing the update to

INTEGRATED NGO MANAGEMENT SYSTEM			
			proceed. The system validates the following: • The Group Name input field is not null or empty. • The Group Name input does not contain only whitespace characters. • Duplicate users are not added to the group's user list when adding members. [ALT]
			Step 7: The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Save" button. • Confirmation modal title ("Confirm group update") • Confirmation message label ○ Displays message: "Are you sure you want to update this group?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 8: The super admin selects the		**Step 9:** The system will then capture the updated Group

INTEGRATED NGO MANAGEMENT SYSTEM			
	"Confirm" button. [ALT]		details and route the data to the corresponding API endpoint using the Angular frontend and group service, which sends the request to the appropriate HTTP PUT endpoint in Visual Studio (GroupController – UpdateGroup). The API endpoint will validate the incoming data and update the existing Group record in the database using Entity Framework Core in MS SQL Server. The following entities are involved in the process: **Group Entity:** • GroupId (PK) • ProjectId (FK) • GroupName • ActiveStatus • DateCreated **UserGroup Entity (Junction Table):** • UserId (PK, FK) • GroupId (PK, FK) The system also updates the UserGroup table to reflect any changes made, including newly added users and removed users. [ALT]

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 10: The system synchronises the updated group data with Firestore to support real-time updates across connected users.
			Step 11: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the group. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a group has been updated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 12: The system will then display a confirmation message indicating that the group has been successfully updated. This is implemented in the Angular frontend using a toaster notification component.

INTEGRATED NGO MANAGEMENT SYSTEM			
			The confirmation message will display the text: "Group updated successfully".
Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and displays an error message within the Edit Group modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 4.		
	ALT-Step 9: The system fails to save the new group and displays a toaster message: "Failed to update group". Terminate use case.		
Conclusion:	The updated group's details have been saved onto the system, and a confirmation message is shown.		
Post-condition:	The system saves the changes to the Group and UserGroup tables.		
Business Rules:	• Only super admin or users given appropriate permissions can update a group's details.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 39: Update Group - Technical Use Case Narrative

2.4.4 Deactivate Group

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Group</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.4</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin deactivates an existing group within the system. The system displays the selected group and allows the Super Admin to select deactivate. The group is then soft deleted by changing the status to inactive, to ensure data integrity and historical tracking. The system will no longer show the group in the active group listings. The use case ends when the group has been successfully deactivated and is no longer visible as an active group within the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to deactivate a group on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to deactivate a group on the system.</td><td></td><td>Step 2: The system invokes UC 4.2 (View Groups) to display a pop-up showing all the groups' details. This is done using Angular frontend and the group service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			Visual Studio. (Group Controller)
	Step 3: The super admin selects the "Deactivate" button on a specific group. [ALT]		**Step 4:** The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Deactivate" button. • Confirmation modal title ("Confirm group deactivation") • Confirmation message label ○ Displays message: "Are you sure you want to deactivate this group?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system will then deactivate the selected group by routing the action through the Angular frontend and group service, which sends the request to the appropriate HTTP DELETE endpoint in Visual Studio (GroupController – DeactivateGroup). The API endpoint will validate the group and update its status in the database using Entity Framework Core in MS SQL

INTEGRATED NGO MANAGEMENT SYSTEM			
			Server. This ensures that the group is soft deleted. The following entity is involved in the process: **Group Entity:** • GroupId (PK) • ProjectId (FK) • GroupName • ActiveStatus • DateCreated The deactivated group is then no longer displayed in the active group listings. [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the group. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a group has been deactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK)

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Timestamp • LogEntry
			Step 8: The system will then display a confirmation message indicating that the group has been successfully deactivated. This is implemented in the Angular frontend using a toaster notification component. The confirmation message will display the text: "Group deactivated successfully".
Alternate courses:	**ALT-Step 3:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 5: The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 2.		
	ALT-Step 6: The system fails to deactivate the group and displays a toaster message: "Failed to deactivate group". Terminate use case.		
Conclusion:	The group has been deactivated on the system, and a confirmation message is shown.		
Post-condition:	The system sets the group's status to inactive in the Group table and the group is then no longer displayed in the active group listings.		
Business Rules:	• Only super admin or users given appropriate permissions can deactivate a group.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 40: Deactivate Group - Technical Use Case Narrative

2.4.5 Create Skill

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Skill</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.5</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin creates a new skill within the system. The system allows the user to enter the skill’s name and description, validates the input, and requests confirmation before saving the skill. Once confirmed, the system sends the data to the backend API endpoint, where the skill is created in the database. The new skill is then displayed on the Skills page as a card, and a confirmation message is shown.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to add a new skill to the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to add a new skill to the system.</td><td></td><td>Step 2: The system invokes UC 4.6 (View Skills) to display all the current skills. This is done using the Angular frontend and skills service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio</td></tr>
</table>

			(SkillController). The retrieved skills are then displayed as skill cards on the Skills page.
	Step 3: The super admin selects the "Add Skill" button.		**Step 4:** The system will then load and display the Create Skill Angular Modal with the following components: • Create Skill modal title • Skill name section: ○ Skill name label and input field ○ Placeholder text: "Skill Name" • Skill description section: ○ Skill description label and input field ○ Placeholder text: "Skill Description" • Action buttons section: ○ Create button (Disabled if the skill name or description is empty) ○ Cancel button
	Step 5: The super admin enters the skill's details and		**Step 6:** The system validates the entered details using Angular form validation and

	then selects the "Create" button. [ALT]		frontend checks. The system validates the following: • The Skill Name input field is not null or empty. • The Skill Name input does not contain only whitespace characters. • The Skill Description input field is not null or empty. • The Skill Description input does not contain only whitespace characters. [ALT]
			Step 7: The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Create" button. • Confirmation modal title ("Confirm skill creation") • Confirmation message label ○ Displays message: "Are you sure you want to create this skill?" • Action buttons: ○ Confirm button

			○ Cancel button
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system will then capture the new Skill's details and route the data to the corresponding API endpoints using the Angular frontend and skills service, which sends the request to the appropriate HTTP POST endpoint in Visual Studio (SkillController – Create). The API endpoint will validate the incoming data and create the new Skill in the database using Entity Framework Core in MS SQL Server. The following entity is involved in the process: **Skill Entity:** • SkillId (PK) • SkillName • SkillDescription • IsActive [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the new skill. The system will store information such as the

			authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a new skill has been created. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system will then display a confirmation message indicating that the skill has been successfully created. This is implemented in the Angular frontend using a toaster notification component. The confirmation message will display the text: "Skill created successfully".
Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and displays an error message within the Create Skill modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 4.		
	ALT-Step 9: The system fails to save the new skill and displays a toaster message: "Failed to save new skill". Terminate use case.		

Conclusion:	The new skill's details have been saved onto the system, and a confirmation message is shown.
Post-condition:	The system saves the new skill onto the Skills table.
Business Rules:	• Only super admin or users given appropriate permissions can add a new skill.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 41: Create Skill - Technical Use Case Narrative

2.4.6 View Skills

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Skills</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.6</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes when an admin views the details of all the skill within the system. The system displays the skills as cards containing the skill’s name and description. The use case ends when the admin sees all the skills displays as cards.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The admin wants to view the skills on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The admin wants to view the skills on the system.</td><td></td><td></td></tr>
<tr><td>Step 2: The admin hovers the “Admin” tab and then selects “Skills”. [ALT]</td><td></td><td>Step 3: The system sends the request through the Angular frontend and skills service to the relevant HTTP GET API endpoint in Visual Studio (SkillController – GetAll), which retrieves all active</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			skills from the Skills table in the database. **Skill Entity:** • SkillId (PK) • SkillName • SkillDescription • IsActive Only active skills are shown.
			Step 4: The system will then display the Skills page using the Angular frontend, showing all retrieved skills as interactive skill cards based on the data returned from the API. • The Skills page includes the following components: • Page title (“Skills”) • Action section: ○ Add Skill button (navigates to Create Skill modal) • Skills display section: ○ Skills are displayed as cards in a grid layout ○ Each skill card displays:

INTEGRATED NGO MANAGEMENT SYSTEM			
			▪ Skill Name ▪ Skill Description • Interaction behaviour: • Each skill card is clickable [ALT]
	Step 5: The admin can view the skills.		
Alternate courses:	**ALT-Step 2:** User is on mobile and selects the menu icon. Go to step 3.		
	ALT-Step 4: The system does not find any active skills. Terminate use case		
Conclusion:	All skills' details are displayed in cards.		
Post-condition:	The system displays all the current skills in cards.		
Business Rules:	• Only super admins or users given appropriate permissions can view the skills.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 42: View Skills - Technical Use Case Narrative

2.4.7 View Skill

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Skill</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.7</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes when an admin views the details of a selected skill within the system. The admin will select a skill card, and it will open a modal. From this modal, the user can view full skill information and access actions such as editing or deactivating the skill. The use case ends when the admin can view the skill's details.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The admin wants to view the specific skill’s details on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The admin wants to view the specific skill’s details on the system.</td><td></td><td>Step 2: The system invokes UC 4.6 (View Skills) to display all the current skills. This is done using the Angular frontend and skills service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (SkillController). The retrieved skills are then</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			displayed as skill cards on the Skills page.
	Step 3: The admin selects a skills card.		**Step 4:** The system retrieves the selected skill details through the skills service in the Angular frontend. The service sends a request via Angular HTTP Client to the HTTP GET endpoint in Visual Studio (SkillController – Get), which retrieves the skill based on the selected SkillId. The API then retrieves data from the following table: **Skill Entity:** • SkillId (PK) • SkillName • SkillDescription • IsActive
			Step 5: The system displays a View Skill Modal on the Angular frontend containing the selected skill details. The modal is triggered when a skill card is selected. The view modal has the following components: • Modal heading: ○ Skill name • Skill details section: ○ Displays the skill description

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Action buttons section: ○ Edit button ○ Deactivate button ○ Close button
	Step 6: The admin can view the skill.		
Alternate courses:	None		
Conclusion:	The system displays a modal showing the skill's details.		
Post-condition:	The skill's details are displayed.		
Business Rules:	• Only super admins or users given appropriate permissions can view a skill.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 43: View Skill - Technical Use Case Narrative

2.4.8 Update Skill

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Update Skill</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.8</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin updates an existing skill in the system. The system allows the user to modify the skill’s name and description on an edit modal. Once the user confirms the changes, the system validates the input and updates the skill in the database. The updated skill information is then reflected on the skills page.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to update a skill’s details on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to update a skill’s details on the system.</td><td></td><td>Step 2: The system invokes UC 4.7 (View Skill) to display the selected skill details in a pop-up modal. This is done using the Angular frontend and skills service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (SkillController). The retrieved skill details are then used to populate the edit form within the modal.</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	Step 3: The super admin selects the "Edit" button.		**Step 4:** The system displays the Edit Skill Modal with the selected skill's details pre-populated and editable. The modal contains the following components: • Edit Skill modal title • Skill name section: ○ Input field ○ Pre-populated with the selected skill's current name • Skill description section: ○ Input field ○ Pre-populated with the selected skill's current description • Action buttons section: ○ Save button ○ Cancel button
	Step 5: The super admin updates the skill's details and selects the "Save" button. [ALT]		**Step 6:** The system validates the updated details using Angular form validation and frontend checks. The system validates the following: • The Skill Name input field is not null or empty. • The Skill Name input does not contain only whitespace characters. • The Skill Description input field is not null or empty.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• The Skill Description input does not contain only whitespace characters. [ALT]
			Step 7: The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Save" button. • Confirmation modal title ("Confirm skill update") • Confirmation message label ○ Displays message: "Are you sure you want to update this skill?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system will then capture the updated Skill details and route the data to the corresponding API endpoint using the Angular frontend and Skills service, which sends the request to the appropriate HTTP PUT endpoint in Visual Studio (SkillController – Update). The API endpoint will validate the incoming data and update the existing Skill record in the database using Entity Framework Core in MS SQL Server.

INTEGRATED NGO MANAGEMENT SYSTEM			
			The following entity is involved in the process: **Skill Entity:** • SkillId (PK) • SkillName • SkillDescription • IsActive [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the group. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a skill has been updated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system will then display a confirmation message indicating that the skill has been successfully updated. This is implemented in the Angular

INTEGRATED NGO MANAGEMENT SYSTEM			
			frontend using a toaster notification component. The confirmation message will display the text: "Skill updated successfully".
Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and displays an error message within the Edit Skill modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 4.		
	ALT-Step 9: The system fails to save the new skill and displays a toaster message: "Failed to update skill". Terminate use case.		
Conclusion:	The updated skill's details have been saved onto the system, and a confirmation message is shown.		
Post-condition:	The system saves the updated skill's details onto the Skills table.		
Business Rules:	• Only super admin or users given appropriate permissions can update a skill.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 44: Update Skill - Technical Use Case Narrative

2.4.9 Deactivate Skill

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Skill</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">4.9</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a Super Admin deactivates an existing skill in the system. The system allows the Super Admin to select deactivate on a skill and will then confirm the deactivation. Once confirmed, the system updates the skill’s status to inactive in the database, which will remove it from the active skills list. The use case ends when the skill is no longer visible within the active skill listings.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to deactivate a skill on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to deactivate a skill on the system.</td><td></td><td>Step 2: The system invokes UC 4.7 (View Skill) to display the selected skill details in a pop-up modal. This is done using the Angular frontend and skills service, which routes the request via Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (SkillController).</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	Step 3: The super admin selects the "Deactivate" button. [ALT]		**Step 4:** The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the "Deactivate" button. • Confirmation modal title ("Confirm skill deactivation") • Confirmation message label ○ Displays message: "Are you sure you want to deactivate this skill?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system will then deactivate the selected skill by routing the action through the Angular frontend and skills service, which sends the request to the appropriate HTTP DELETE endpoint in Visual Studio (SkillController – Deactivate). The API endpoint will validate the skill and update its status in the database using Entity Framework Core in MS SQL Server. This ensures that the skill is soft deleted. The following entity is involved in the process: **Skill Entity:** • SkillId (PK) • SkillName • SkillDescription

INTEGRATED NGO MANAGEMENT SYSTEM			
			• IsActive The deactivated skill is then no longer displayed in the active skills listings. [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the skill. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a skill has been deactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system will then display a confirmation message indicating that the skill has been successfully deactivated. This is implemented in the Angular frontend using a toaster notification component. The confirmation message will display the text: “Skill deactivated successfully”.
Alternate courses:	**ALT-Step 3:** The super admin changes their mind and closes the pop-up by selecting the “Cancel” button. Terminate use case.		
	ALT-Step 5:		

INTEGRATED NGO MANAGEMENT SYSTEM	
	The super admin does not confirm the changes by selecting the "Cancel" button. Go back to step 2.
	ALT-Step 6: The system fails to deactivate the skill and displays a toaster message: "Failed to deactivate skill". Terminate use case.
Conclusion:	The active skill has been deactivated, and a confirmation message is shown.
Post-condition:	The system updates the skill's status to inactive in the Skill table.
Business Rules:	• Only super admin or users given appropriate permissions can deactivate a skill.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 45: Deactivate Skill - Technical Use Case Narrative

2.5 Warehouse Subsystem Use Case Narratives

2.5.1 Create Warehouse

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Create Warehouse</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>5.1</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin registers a new warehouse on the Inventory Management page. The Admin opens the “Add Warehouse” form on the Angular frontend and enters the warehouse’s name, location, capacity, and security level. Once the form passes validation, the system displays a confirmation prompt. On confirmation, the Angular WarehouseService sends the details to the corresponding “.NET 8” API endpoint (“POST api/Warehouse”). The backend validates the incoming data and, using Entity Framework Core, inserts a new record with an active status. The use case concludes with the frontend refreshing the warehouse list and displaying a success notification.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin must be authenticated and viewing the Warehouses page.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin requests to add a new warehouse by selecting the “Add Warehouse” button.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Admin navigates to the</td><td></td><td>Step 2: The system loads the Warehouses</td></tr>
</table>

	Inventory Management page and selects the "Warehouses" tab.		screen, displaying a grid of existing warehouses and an "Add Warehouse" button.
	Step 3: Admin selects "Add Warehouse".		**Step 4:** The system opens a form containing fields for Name, Location, Capacity, and Security Level, with the "Create" button disabled until all fields are valid.
	Step 5: Admin completes the form and selects "Create". [ALT]		**Step 6:** The system checks that Name and Location are not empty and that Capacity is a positive number, then displays a confirmation prompt: "Are you sure you want to create this Warehouse?"
	Step 7: Admin selects "Confirm".		**Step 8:** The WarehouseService sends the warehouse details to "POST api/Warehouse". The controller validates the request and Entity Framework Core inserts a new record into the following entity: **Warehouse Entity:** • WarehouseId (PK) • Name • Location • Size

			• SecurityLevel • Status
			Step 9: The new warehouse is saved with its Status set to active by default. The API responds with an HTTP 201 Created status, and the frontend refreshes the warehouse grid and displays a success notification.
Alternate courses:	**[ALT] Step 5**: Admin leaves a required field empty or enters an invalid capacity. The system displays an inline error message and keeps the form open.		
	[ALT] Step 8a: The API returns an HTTP 400 response due to invalid data. The frontend displays "Please check the information entered" and reopens the form.		
	[ALT] Step 8b: The API returns an HTTP 403 response because the Admin lacks permission. The frontend displays "You do not have permission to add warehouses."		
	[ALT] Step 8c: The API returns an HTTP 500 response due to a server error. The frontend displays "System error, please try again."		
Conclusion:	The use case concludes when the new warehouse is successfully saved to the database and appears in the warehouse grid.		
Post-condition:	A new record exists in the Warehouse entity with an automatically generated WarehouseId and an active Status. The warehouse is now available for inventory and stock take operations.		
Business rules:	• Name and Location are required fields. • Capacity must be a positive number. • A newly created warehouse is always set to active.		
Implementation constraints and specifications:	• Entity Framework Core generates the WarehouseId automatically when the record is saved.		
Assumptions:	• The Admin provides accurate warehouse details. There is currently no check preventing duplicate warehouse names.		

Open issues:	• No server-side check exists to prevent duplicate warehouse name

Table 46: Create Warehouse - Technical Use Case Narrative

2.5.2 View Warehouse

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Warehouse</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>5.2</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin views the details of a specific warehouse. From the Warehouses page, the Admin selects a warehouse card, and the system retrieves the warehouse’s details from the database and displays them in a details window, along with buttons to edit, deactivate, or reactivate the warehouse depending on its status.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">The Admin is authenticated and the selected warehouse exists in the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects a warehouse card from the Warehouses grid.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: Admin navigates to the Inventory Management page and selects the “Warehouses” tab.</td><td></td><td>Step 2: The WarehousesComponent calls “GET api/Warehouse”, which retrieves all warehouse records from the Warehouse entity. The system displays each warehouse as a card showing its name, location, capacity,</td></tr>
</table>

			security level, and an Active/Inactive status label.
	Step 3: Admin selects a warehouse card.		**Step 4:** The system calls "GET api/Warehouse/{id}" to retrieve the selected warehouse's details from the Warehouse entity. [ALT]
			Step 5: The system opens a details window showing the warehouse's Name, Location, Capacity, Security Level, and Status, along with "Edit", "Deactivate" (if active) or "Reactivate" (if inactive), and "Close" buttons.
Alternate courses:	**[ALT] Step 4:** The warehouse cannot be found. The system does not open the details window.		
Conclusion:	The use case concludes when the selected warehouse's details are displayed to the Admin.		
Post-condition:	No data is modified. The warehouse details remain visible until the Admin closes the window.		
Business rules:	• Only authenticated users can view warehouse details. • Active warehouses display a "Deactivate" option; inactive warehouses display a "Reactivate" option.		
Implementation constraints and specifications:	• "GET api/Warehouse/{id}" returns the warehouse's details through a read-only query, since no data is modified.		
Assumptions:	• The warehouse data shown reflects its state at the time it was retrieved.		
Open issues:	• No error message is shown if the warehouse cannot be found.		

Table 47: View Warehouse - Technical Use Case Narrative

2.5.3 Update Warehouse

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Stephan Bredenhann</td><td>Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td colspan="2">Update Warehouse</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">5.3</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin edits an existing warehouse’s details. From the warehouse details window, the Admin selects “Edit”, updates the pre-filled form, and confirms the change. The system then updates the corresponding record in the database. The warehouse’s active or inactive status is not affected by this use case, since that is handled separately by the deactivate and reactivate use cases.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated, the warehouse exists, and its details window is open.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Edit” in the warehouse details window.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Admin opens a warehouse’s details window (see Use Case 5.2) and selects “Edit”.</td><td></td><td>Step 2: The system closes the details window and opens an update form pre-filled with the warehouse’s current Name, Location, Capacity, and Security Level.</td></tr>
</table>

	Step 3: Admin updates the fields and selects "Update". [ALT]		**Step 4:** The system validates that Name and Location are not empty and Capacity is a positive number, then displays a confirmation prompt: "Are you sure you want to update this Warehouse?"
	Step 5: Admin selects "Confirm".		**Step 6:** The WarehouseService sends the updated details to "PUT api/Warehouse/{id}". The controller checks that the warehouse exists, and Entity Framework Core updates the Name, Location, Size, and SecurityLevel fields on the corresponding Warehouse record. The WarehouseId and Status fields are left unchanged. [ALT]
			Step 7: The API responds with an HTTP 204 status. The frontend refreshes the warehouse grid and displays a success notification.
Alternate courses:	**[ALT] Step 3:** Admin clears a required field or enters an invalid capacity. The system displays an inline error and keeps the form open.		
	[ALT] Step 6a: The warehouse cannot be found. The system displays a generic error message.		
	[ALT] Step 6b:		

	The API returns an HTTP 400 response due to invalid data. The system displays a generic error message.
Conclusion:	The use case concludes when the warehouse's details are successfully updated and reflected in the warehouse grid.
Post-condition:	The warehouse's Name, Location, Size, and SecurityLevel are updated. The WarehouseId and Status remain unchanged.
Business rules:	• Only authenticated users can update warehouse details. • The Status field cannot be changed through this use case (see Use Cases 5.4 and 5.5).
Implementation constraints and specifications:	• "PUT api/Warehouse/{id}" requires the warehouse ID in the request body to match the ID in the URL. • The update is applied directly to the tracked Warehouse entity and saved to the database.
Assumptions:	• None.
Open issues:	• There is no protection against two Admins updating the same warehouse at the same time.

Table 48: Update Warehouse - Technical Use Case Narrative

2.5.4 Reactivate Warehouse

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Reactivate Warehouse</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>5.4</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin restores an inactive warehouse to active status. From the details window of an inactive warehouse, the Admin selects “Reactivate” and confirms the action. The system updates the warehouse’s Status field to active.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">The Admin is authenticated and the warehouse’s Status is currently inactive.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Reactivate” in the details window of an inactive warehouse.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Admin opens the details window of an inactive warehouse and selects “Reactivate”.</td><td></td><td>Step 2: The system displays a confirmation prompt: “Are you sure you want to reactivate this Warehouse?”</td></tr>
<tr><td>Step 3: Admin selects “Confirm”.</td><td></td><td>Step 4: The WarehouseService sends a request to “PUT api/Warehouse/{id}/reactivate”.</td></tr>
</table>

			Entity Framework Core updates the Status field on the corresponding Warehouse record to active, leaving all other fields unchanged. [ALT]
			Step 5: The API responds with an HTTP 204 status. The frontend refreshes the warehouse grid and displays a success notification.
Alternate courses:	**[ALT] Step 3:** Admin selects "Cancel". The prompt closes and no changes are made.		
	[ALT] Step 4: The warehouse cannot be found. The system displays a generic error message.		
Conclusion:	The use case concludes when the warehouse's Status is set to active.		
Post-condition:	The warehouse's Status is active. Its card displays a green "Active" label, and its details window now shows a "Deactivate" option.		
Business rules:	• Only authenticated users can reactivate a warehouse.		
Implementation constraints and specifications:	• "PUT api/Warehouse/{id}/reactivate" requires no request body.		
Assumptions:	• None.		
Open issues:	• None.		

Table 49: Reactivate Warehouse -Technical Use Case Narrative

2.5.5 Deactivate Warehouse

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Deactivate Warehouse</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>5.5</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">User</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin deactivates an active warehouse, removing it from future inventory and stock take activity while keeping its historical data intact. From the details window, the Admin selects “Deactivate” and confirms the action. The system updates the warehouse’s Status field to inactive. Note that the system does not currently check whether the warehouse still holds inventory before it is deactivated, even though the original design specification calls for this check.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated and the warehouse’s Status is currently active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Deactivate” in the details window of an active warehouse.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Admin opens the details window of an active warehouse and selects “Deactivate”.</td><td></td><td>Step 2: The system displays a confirmation prompt: “Are you sure you want to deactivate this Warehouse?”</td></tr>
</table>

	Step 3: Admin selects "Confirm".		**Step 4:** The WarehouseService sends a request to "PUT api/Warehouse/{id}/deactivate". Entity Framework Core updates the Status field on the corresponding Warehouse record to inactive, leaving all other fields unchanged. No check is performed against existing Inventory or Stock Take records. [ALT]
			Step 5: The API responds with an HTTP 204 status. The frontend refreshes the warehouse grid and displays a success notification.
Alternate courses:	**[ALT] Step 3:** Admin selects "Cancel". The prompt closes and no changes are made.		
	[ALT] Step 4: The warehouse cannot be found. The system displays a generic error message.		
Conclusion:	The use case concludes when the warehouse's Status is set to inactive.		
Post-condition:	The warehouse's Status is inactive. Its card displays a red "Inactive" label, and its details window now shows a "Reactivate" option. Existing inventory records that reference the warehouse are preserved, since deletion is restricted at the database level.		
Business rules:	• Only authenticated users can deactivate a warehouse. • Deactivation is a soft delete; the warehouse record itself is never removed.		
Implementation constraints and specifications:	• "PUT api/Warehouse/{id}/deactivate" requires no request body.		
Assumptions:	• None.		
Open issues:	• None.		

Table 50: Deactivate Warehouse - Technical Use Case Narrative

2.5.6 Inventory Stock Take

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Stephan Bredenhann	**Date:** 13 July 2026 **Version:** 1.0	
USE CASE NAME:	Inventory Stock Take	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	5.6		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	User		
Description:	The Super Admin performs a physical stock take against a single warehouse. The Super Admin opens the Stock Take page, selects a warehouse, and begins a stock take. The system takes a snapshot of every inventory item in that warehouse whose status is Available and creates one stock take line per item, each with a system quantity of 1 and a counted quantity of 0. The Super Admin then walks the warehouse floor, records the physical count for each line, and submits the stock take. On submission the system stores the counts, calculates the difference for each line, marks the stock take as Completed, and automatically writes off every inventory item that was counted as 0, so that the system inventory is adjusted to match the physical count.		
Pre-condition:	• Super Admin is authenticated and holds stock take permissions. At least one active warehouse exists in the system.		
Trigger:	Super Admin clicks "New Stock Take" on the Stock Take page.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Super Admin navigates to /inventory-management and opens the "Stock Take" tab.		**Step 2:** System renders the StockTakeComponent and calls GET api/StockTake. The StockTakeController returns a list of StockTakeListDto records ordered by DateConducted

			descending, then StockTakeId descending, and the grid displays the stock take number, warehouse name, date, status and line count.
	Step 3: Super Admin clicks "New Stock Take".		**Step 4:** System opens the new stock take view and calls GET api/Warehouse to populate the warehouse dropdown. The "Begin Stock Take" button stays disabled until a warehouse is selected.
	Step 5: Super Admin selects a warehouse from the dropdown.		**Step 6:** System calls GET api/Inventory?warehouseId={id} and displays the current inventory of the selected warehouse as a read-only preview, together with the total item count, so that the Super Admin can see what is about to be counted.
	Step 7: Super Admin clicks "Begin Stock Take". [ALT]		**Step 8:** StockTakeService sends POST api/StockTake with a CreateStockTakeRequest containing the WarehouseId. The StockTakeController calls WarehouseExists(warehouseId) and returns HTTP 400 "Warehouse does not exist." if the warehouse cannot be found.
			Step 9: On success, the StockTakeRepository reads every Inventory row where WarehouseId matches the request and Status = Available (1). Items with status Allocated (2) or WrittenOff (3) are excluded from the snapshot. The repository then persists a new

			stock take and one line per available item. **StockTake Entity:** • StockTakeId (PK) • WarehouseId (FK) • DateConducted • Status **StockTakeLine Entity:** • StockTakeLineId (PK) • StockTakeId (FK) • InventoryItemId (FK) • SystemQuantity • CountedQuantity
			Step 10: System returns HTTP 201 Created with a StockTakeDetailsDto and a Location header pointing at GET api/StockTake/{newId}. The frontend renders the count entry screen listing every line with its SKU, item name and system quantity.
	Step 11: Super Admin walks the warehouse floor and enters the physical count for each line, then clicks "Submit Stock Take". [ALT]		**Step 12:** System shows a confirmation dialog summarising the total number of items, the number counted, the number missing, and the list of items that will be written off, with the warning that the action cannot be undone.
	Step 13: Super Admin clicks "Confirm & Submit".		**Step 14:** StockTakeService sends PUT api/StockTake/{id}/submit with a SubmitStockTakeRequest containing the StockTakeId and every line with its counted quantity. The controller checks that the id in the URL matches the StockTakeId in the body and returns HTTP 400 "StockTakeId mismatch." if they differ. [ALT]

			Step 15: The repository loads the stock take with its lines using Include, updates the CountedQuantity of each matching line, and sets the stock take status to Completed. **StockTakeLine Entity:** • CountedQuantity **StockTake Entity:** • Status
			Step 16: System applies the automatic inventory adjustment. Every line whose CountedQuantity is 0 is treated as a missing item, and the matching inventory record has its status changed to WrittenOff (3). All changes are saved in a single SaveChangesAsync() call. **Inventory Entity:** • Status
			Step 17: System returns HTTP 200 OK with the updated StockTakeDetailsDto. The frontend renders the discrepancy report, showing the system quantity, counted quantity and the difference (CountedQuantity - SystemQuantity) for each line, a totals row, and a warning listing how many items were automatically written off.
Alternate courses:	**[ALT] Step 7:** The selected warehouse no longer exists. The API returns HTTP 400 "Warehouse does not exist." and the frontend shows "That warehouse could not be found." The stock take is not created.		
	[ALT] Step 9:		

	The warehouse contains no items with status Available. The stock take is still created but with zero lines, and the count entry screen shows an empty list with the message "There is nothing to count in this warehouse."
	[ALT] Step 11: The Super Admin leaves lines at a count of 0 without physically checking them. The confirmation dialog lists every one of those items as a pending write-off so that the mistake is visible before the submission is confirmed.
	[ALT] Step 14a: The id in the URL does not match the StockTakeId in the request body. The API returns HTTP 400 "StockTakeId mismatch." and the frontend shows "Please check the information entered."
	[ALT] Step 14b: The stock take no longer exists. The API returns HTTP 404 and the frontend show "That stock take could not be found."
	[ALT] Step 14c: The API returns HTTP 403. The frontend shows "You do not have permission to perform a stock take."
	[ALT] Step 14d: The API returns HTTP 500. The frontend shows "System error, please try again." and the captured counts remain on screen so that the submission can be retried.
Conclusion:	The stock take is stored in the [StockTakes] table with a status of Completed, every line in [StockTakeLines] carries the counted quantity entered by the Super Admin, and every inventory item that could not be found has been written off. The discrepancy report is displayed, and the stock take appears at the top of the stock take list.
Post-condition:	A new row exists in [StockTakes] with Status = 'Completed', one row per counted item exists in [StockTakeLines], and every [Inventory] row whose CountedQuantity was 0 now has Status = WrittenOff (3). Written-off items are excluded from future stock takes and from allocation.
Business Rules:	• Only a user with stock take permissions may create or submit a stock take. • A stock take can only be created against an existing warehouse. • Only inventory items with status Available (1) are included in a stock take; Allocated and written-off items are excluded. • A stock takes moves through two statuses only: In Progress, then Completed. • Each inventory record represents a single unit, so SystemQuantity is always 1.

	• The difference on a line is calculated as CountedQuantity minus SystemQuantity. • Any line submitted with a counted quantity of 0 causes the matching inventory item to be written off automatically.
Implementation constraints and specifications:	• GET api/StockTake returns List<StockTakeListDto> and uses AsNoTracking() for read-only performance. • GET api/StockTake/{id} returns StockTakeDetailsDto and responds with HTTP 404 when the stock take does not exist. • POST api/StockTake accepts CreateStockTakeRequest (WarehouseId) and responds with HTTP 201 Created. • PUT api/StockTake/{id}/submit accepts SubmitStockTakeRequest (StockTakeId, Lines) and responds with HTTP 200 OK. • StockTakeLine has a cascade delete on StockTakeId and a restrict delete on InventoryItemId, so counted inventory cannot be deleted while a stock take references it. • The Difference field is calculated in the projection and is not stored in the database. • StockTakeRepository and StockTakeService are registered as scoped services.
Assumptions:	• Super Admin counts the warehouse accurately and submits every line in one request; the physical count is trusted over the system record.
Open issues:	• None • The [Authorize] attributes on the StockTakeController are still commented out, so the endpoints are currently unprotected. • Stock take creation and submission are not written to the audit log, so there is no record of who performed the count. • A submitted stock take cannot be reopened or corrected, and a second submission silently overwrites the first.

Table 51: Inventory Stock Take - Technical Use Case Narrative

2.6 Vendor Subsystem Use Case Narratives

2.6.1 Create Vendor

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Create Vendor	**USE CASE TYPE**	
Use case id:	6.1	Business Requirements: □	
Priority:	High	System Analysis: □	
Source:	EPI-USE	System Design: ✓	
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the system flow where an authorized user registers a new vendor profile within the application. The user captures corporate metrics, type parameters, contact paths, and an image profile asset into an open Angular Reactive Form Group layout interface. Upon clicking save, the frontend interface verifies control validity and dispatches an asynchronous multi-part HTTP POST call payload to the corresponding backend .NET 8 API endpoint router path (POST api/Vendor/create) carrying the text models along with base64 data strings representing the image file. The backend controller binds incoming parameters to an explicit database context tracking instance, executes validation algorithm loops, and commits changes via Entity Framework Core. The use case finishes once the frontend updates directory data grids, closes form panels, and fires a green confirmation toast notification.		
Pre-condition:	• The executing user must be authenticated and hold appropriate vendor registry creation and system management claims.		
Trigger:	The user requests to register a new vendor within the application by clicking the "Add New Vendor" action button trigger component interface link.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

	Step 1: The user requests to register a new vendor within the application by clicking the "Add New Vendor" action button trigger component interface link.		**Step 2:** The Angular framework handles view compilation logic, loading the dedicated Vendor Form template context into the active workspace slot.
		Step 3: The client interface generates and renders an open, interactive form pane display presenting a structured Angular Reactive Form Group containing empty text boxes, option dropdowns, and file uploader utilities targeting corporate company name, vendor classification type selection, performance metrics score, contact email string details, telephone digits, credit facility caps,	

		physical street address lines, and a profile image stream drop-zone.	
	Step 4: The user populates the relevant descriptive data input targets, links a business type category classification, attaches an initial performance score, triggers an image upload stream file select, and hits the primary "Save Vendor" record submission controller [ALT]		**Step 5:** The client-side framework evaluates active form control validation states. The Angular VendorService builds and dispatches an asynchronous multi-part HTTP POST call payload targeting backend routing path endpoints api/Vendor/create passing the text models along with base64 data strings representing the image file. The backend .NET 8 service intercepts the package payload context. The runtime builds a database connection via the ApplicationDbContext and queries structural repository indices to verify that the target vendor registration numbers or email addresses do not trigger duplication violations.
			Step 6: The API executes model binding constraint evaluations on the application server layers. The system checks data structure parameters

			against backend fields (checking mandatory items, email string notation properties, numeric limits, and checking telephone formats).
			Step 7: Upon successful validation passes across all system parameters, the backend instantiates a new record row entry instance for the Vendor data model map, appends automated transaction timestamp tracking properties, and issues an asynchronous commit call (SaveChangesAsync).
			Step 8: The application server responds with an HTTP 201 Created action success block stream. The frontend catches the transaction data response, inserts the newly returned vendor profile instance into its master directory data display grids, closes the form pane interface drawers, and displays a green confirmation toast notice layout stating: "Vendor record successfully added to global system directories."
Alternate courses:	**[ALT] Step 4:**		

	The user clicks the "Discard" or "Exit" interface option inside the form panel framework drawer layout. The frontend interface stops outbound JSON payload packaging routines, releases active form variables from memory buffers, prevents network data calls to the server, and closes down active panels.
Conclusion:	The use case concludes when the system successfully validates registration parameters, establishes tracking rows inside the database backend tables, and updates global directory data grids.
Post-condition:	• A new vendor model row profile is permanently written to the database structure, enabling its selection across lookup picklists and project procurement transaction modules.
Business rules:	• Every vendor profile record row must contain a completely unique business contact email string notation to prevent directory mapping overlaps. • New vendor profiles must bind to non-negative initial performance tracking score variables.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 52: Create Vendor - Technical Use Case Narrative

2.6.2 View Vendor

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	View Vendor	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.2		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case describes how the system retrieves and displays a comprehensive, standalone read-only view of a selected vendor entity profile line item. When a user selects a record row, the frontend evaluates routing parameters and dispatches an asynchronous HTTP GET request to the matching .NET 8 API endpoint (GET api/Vendor/details/{id}). The backend controller targets the relational rows via the master ApplicationDbContext, executing optimized select query statement operations that perform automated table joins across linked lookup schemas (classification types, contact address arrays, performance records). The server packages variables into a structured Data Transfer Object (DTO) payload and dispatches an HTTP 200 OK code back to the client thread. The use case ends when the frontend maps incoming metrics directly into template interpolations, rendering data arrays into uneditable screen components.		
Pre-condition:	• The target vendor instance record must currently exist inside the database layer partitions with functional active index mapping nodes.		
Trigger:	The user selects a specific vendor entity profile line-item row within the master directory data table list by clicking its corresponding "View Details" text link or row element.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user selects a specific vendor entity		**Step 2:** The Angular Router engine captures

	profile line-item row within the master directory data table list by clicking its corresponding "View Details" text link or row element.		the user click event stream, evaluates routing templates, and triggers path transitions navigating the layout container over to the dedicated vendor single profile view path, appending the unique ID parameter mapping (/vendors/view/:id).
			Step 3: The profile display component launches its initialization lifecycle hooks (ngOnInit). The underlying Angular VendorService builds and fires an asynchronous HTTP GET transaction targeting the backend server resource URL path parameters mapping api/Vendor/details/{id}. The .NET 8 endpoint processes incoming parameters. The underlying architecture builds database calls via the system ApplicationDbContext layer and runs an optimized, compiled read query select statement (FirstOrDefaultAsync) to isolate rows corresponding to the passed identifier [ALT].
			Step 4: The database engine handles

			automated table joining procedures across associated lookup relational models, fetching tracking variables linked to the primary entity schema including corporate names, telephone parameters, profile images, and historical performance statistics lines.
			Step 5: The backend system architecture transforms the returned raw entities into a structured, lightweight Data Transfer Object (DTO) contract collection and returns an HTTP 200 OK success transmission block to the browser client.
			Step 6: The Angular frontend application intercepts the structured network data. The view component processes the model properties and updates template interpolation states to display complete email paths, ratings, and uploaded imagery files on the interface while locking form inputs into an uneditable, read-only display layout state.

Alternate courses:	**[ALT] Step 3**: The database fetch query evaluates an empty data block stream because the targeted vendor record primary key sequence has been concurrently modified or removed from disk. The backend web API stops the execution loop and delivers an HTTP 404 Not Found error packet. The frontend interceptor captures the status error token, breaks loading states, and displays a red warning dialog stating: "The requested vendor profile record could not be fetched. It may have been deactivated or dropped from system directories."
Conclusion:	The use case concludes cleanly when data streams transfer successfully from backend databases, parsing tracking metrics directly into uneditable, read-only interface blocks.
Post-condition:	• The single vendor entity's descriptive statistics, corporate attributes, and performance evaluations are rendered seamlessly on the user interface grid framework.
Business rules:	• Visibility rules require the system to subject single view detail queries to authorization algorithms ensuring the active user holds clearance claims to browse corporate data repositories.
Implementation constraints and specifications:	• Processing routines should leverage specialized Data Transfer Objects (DTOs) to enforce network transmission boundaries and strip core infrastructure properties from transmission payloads.
Assumptions:	• None
Open issues:	• None

Table 53: View Vendor - Technical Use Case Narrative

2.6.3 View Vendors

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Amahle Khoza</td><td>Date: 12 June 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td colspan="2">View Vendors</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">6.3</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">• Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Admin</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the system extracts and lists all registered vendor tracking profiles from backend database layers. Upon loading the master portal module layout, the client application framework initializes initialization lifecycle code hooks, triggering the internal VendorService to dispatch an asynchronous HTTP GET request transaction to the corresponding server routing URL path (GET api/Vendor/list). The .NET 8 backend controller handles incoming request parameters and queries active rows using Entity Framework Core across global vendor datasets, applying sorting configurations dynamically. The server bundles records into an array list data transfer block and returns an HTTP 200 OK response code. The use case finishes once the frontend loops through returned structures via *ngFor row generation commands, rendering rows inside an interactive table grid layout.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be authenticated and maintain active data connectivity streams linking host server environments.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user navigates to the master Vendor Directory portal layout module via the system dashboard link sidebar menu framework.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user accesses the centralized Vendor Management workspace directory</td><td></td><td>Step 2: The Angular client shell captures routing events, updates active layout slots, and</td></tr>
</table>

	layout by clicking on the designated "Vendors" link button option inside the primary sidebar application navigation menu.		initializes the parent Vendor List Component view container framework within the browser window DOM.
			Step 3: The view initialization hooks trigger automated communication sequences. The internal client VendorService builds and dispatches an asynchronous HTTP GET request to the target server API list path route context api/Vendor/list, passing search query filters or status constraint settings. The .NET API controller passes execution commands over to data access contexts. The framework calls into the master ApplicationDbContext instance and prepares an evaluation query string targeted at extracting datasets from global active Vendor tables.
			Step 4: The database instance executes a compiled select query via Entity Framework Core, scanning table rows, sorting records alphabetically by corporate company name parameters, and

			marshaling records into a clean data collection stream.
			Step 5: The application server packs the assembled record items into an array payload model block structure and issues an HTTP 200 OK network success message package back to the client browser app..
	Step 6: The user can adjust dropdown filters or option checkboxes to filter by status (e.g., toggling to view active rows only), which triggers updated compilation data queries.		**Step 7:** The Angular layout engine processes the incoming array block. The framework evaluates template structural looping directives (*ngFor) and loops through the objects to dynamically generate structured table grid rows that present corporate titles, classification indicator pills, and action control asset overlays on screen.
Alternate courses:	None		
Conclusion:	The use case concludes cleanly when serialized collection records compile successfully from databases, updating structural dashboard views to project row grids.		
Post-condition:	• Registered vendor directory data profiles are extracted from backend systems and mapped onto structural interface layouts for query navigation.		
Business rules:	• Security parameters strictly filter dataset records matching security access matrix constraints allocated to the authenticated user profile.		
Implementation constraints and specifications:	• None		

Assumptions:	• None
Open issues:	• None

Table 54: View Vendors - Technical Use Case Narrative

2.6.4 Edit Vendor

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Edit Vendor	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.4		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process where an authenticated operator alters specific descriptive, financial, or contact parameters inside an existing vendor's active registration profile. The user overrides cell properties within an Angular Reactive Form Group layout interface container modal. Upon submission, the client interface runs structural validation checks and routes an asynchronous HTTP PUT network transaction directly to the backend API endpoint (PUT api/Vendor/edit/{id}). The .NET 8 backend references the shared ApplicationDbContext tracker context to isolate the target row entity, runs duplication and constraint verification logic, and flags updates via Entity Framework Core. The use case finishes when mutations write cleanly to relational tables, the editing dialog modal dismisses, and the calling view data layout refreshes.		
Pre-condition:	• The targeted vendor profile instance must currently exist in persistent SQL storage tables with functional, active row mapping indexes, and the operator must possess valid registration write claims.		
Trigger:	The user clicks the specific row context "Edit" action button trigger control asset within the master registry directory table.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user requests to alter specific descriptive		

	parameters inside an existing vendor's registration profile card by clicking the row "Edit" action button trigger.		
		Step 2: The frontend application launches localized editing modal overlay components over the active screen view context, automatically mapping current database structural field parameters into active input boxes via form data control patch value routines.	
	Step 3: The user overrides specific parameter properties within active input controls (modifying phone numbers, expanding credit facility caps, or replacing attachment image files) and clicks the "Save Changes" submit button. [ALT]		**Step 4:** The client application handles validation constraint checks across input properties. The Angular VendorService formats an asynchronous HTTP PUT call dispatched directly to the API routing endpoint template string context api/Vendor/edit/{id}, packing the updated

			payload map. The backend web API intercepts the data transmission stream. The execution thread references the shared ApplicationDbContext context block and passes asynchronous search statements targeting matching primary key vendor identities across relational database indices.
			Step 5: The server backend processes validation verification checks to ensure the target row data item index remains valid, text characters remain inside field type allocations, and modified entry properties do not generate record duplication path collisions.
			Step 6: Upon successful logic validation confirmations, the application context tracks item level changes, overwrites database schema column records with payload parameters, and commits data updates via an asynchronous save update call (SaveChangesAsync).

			Step 7: The .NET API shifts execution pipelines and delivers an HTTP 200 OK response status statement block back to the calling client. The frontend intercepts the success code message, closes down the modal dialogue wrapper overlay panels, reloads view data layouts, and fires a green confirmation toast reading: "Changes committed successfully."
Alternate courses:	**[ALT] Step 3:** The user aborts adjustments and clicks the "Discard" or "Cancel" control option. The frontend interface stops outbound JSON compilation routines, drops all form state alterations from memory buffers, prevents network data calls to the server host, and closes active panels immediately.		
Conclusion:	The use case concludes once all systemic validations clear successfully, updated parameters persist within database disk files and tracking directory layers re-render.		
Post-condition:	• The target vendor instance database row fields are permanently modified with the new configuration parameters, immediately reflecting changes across downstream procurement and accounting workflows.		
Business rules:	• Any modification targeting fundamental financial parameters (such as credit facility caps or bank details) must trigger an automated background audit trail log capturing the operator ID and field delta. • Altered contact email string notations must remain completely unique across all active and inactive vendor registry records.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		

Open issues:	• None

Table 55: Edit Vendor - Technical Use Case Narrative

2.6.5 Reactivate Vendor

<table>
<tr><th colspan="3">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Amahle Khoza</td><td>Date: 12 June 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Reactivate Vendor</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>6.5</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="2">• Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="2">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="2">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="2">Admin</td></tr>
<tr><td>Description:</td><td colspan="2">This use case details the workflow executed when an authorized operator restores a previously suspended or deactivated vendor profile entry back to full system operational availability. The user triggers the action from a deactivated row item context, causing the frontend to display a verification prompt overlay dialog warning framework. Upon confirmation, the client browser application issues an asynchronous HTTP PATCH operations statement directly to the API path parameter route (PATCH api/Vendor/reactivate/{id}). The backend .NET 8 controller handles the arrival command, uses the ApplicationDbContext to locate the target entity row, and executes authorization confirmation checks. The system updates the status flag parameter to true, persists changes to disk, closes the overlay, and pushes a green success confirmation toast message banner.</td></tr>
<tr><td>Pre-condition:</td><td colspan="2">• The targeted vendor record entry must exist inside database partitions with an active status flag currently set to false (IsActiveStatus = false).</td></tr>
<tr><td>Trigger:</td><td colspan="2">The user hits the specific row context "Reactivate" action switch trigger asset control on a deactivated data grid line item.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to restore a previously suspended vendor profile entry by</td><td></td><td></td></tr>
</table>

	hitting the row "Reactivate" contextual action button trigger asset control on a deactivated data item row.		
		Step 2: The client framework launches a warning modal prompt overlay display, showing a descriptive verification question string text layout to the system operator: "Are you sure you want to reactivate vendor [Vendor Name]? It will immediately reappear when creating project assignments."	
	Step 3: The user clicks on the "Confirm Reactivate" action switch button located within the active prompt modal window dialog overlay container. [ALT]		**Step 4:** The frontend interface processes the click event intent. The Angular VendorService builds and dispatches an asynchronous HTTP PATCH operations network statement directly targeting the API url parameter path route

			structure api/Vendor/reactivate/{id}. The .NET backend controller captures the incoming patching command request. The application layer triggers a target data fetch process via the ApplicationDbContext instance to isolate the targeted row record tracking structure inside persistent storage systems.
			Step 5: The system subjects execution calls internal backend authorization validations and database checks to confirm the row entity profile index matches valid registry structures and ensure the operator holds valid workspace claim rights.
			Step 6: Upon verification approvals, the ORM entity state tracker context isolates the explicit status parameter target column, flips the data visibility value flag over to true (IsActiveStatus = true), and persists transaction modifications to persistent database disk layers via an asynchronous statement (SaveChangesAsync).
			Step 7: The web API transfers an HTTP 200 OK network success

			verification payload package. The client browser interceptor handles the code status, updates the data list view layout configuration matrices, closes the dialog prompt modal window context, and pops a green confirmation message alert banner reading: "Vendor profile reactivated successfully."
Alternate courses:	**[ALT] Step 3:** The user decides to cancel reactivation and clicks the "Cancel" or "Dismiss" button control within the prompt layout block container wrapper. The system immediately closes the modal prompt window view context, drops the execution intent, and leaves the database record entry status completely unchanged.		
Conclusion:	The use case concludes when the system completes backend status state validations, successfully updates database table records, closes the active verification overlay dialog wrapper, and refreshes the master data grid cache.		
Post-condition:	• The target vendor entity's data visibility flag is permanently flipped to true (IsActiveStatus = true), making the profile immediately visible across project procurement lookup picklists.		
Business rules:	• Status elevation operations are restricted to operators holding valid administrative configuration or high-level procurement workspace claims.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 56: Reactivate Vendor - Technical Use Case Narrative

2.6.6 Deactivate Vendor

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Deactivate Vendor	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.6		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the technical steps executed to suspend a vendor profile line-item row, restricting it from being selected for future project workflows without deleting its historic transactional tracking data. The user triggers the action from the row context. The frontend interrupts workflows to mount a confirmation warning alert overlay modal block framework. Upon user affirmation, the Angular service layer issues an asynchronous HTTP PATCH request to the backend controller path (PATCH api/Vendor/deactivate/{id}). The .NET 8 API handles the tracking token, verifies database row integrity via the master ApplicationDbContext, and evaluates active relational associations. If validations pass, the runtime switches the entity status property flag over to false. The use case finishes once updates save to disk, the modal disappears from the DOM view, and a green success toast fires.		
Pre-condition:	• The targeted vendor profile record must exist inside the database context frames with a functional active index state matching true (IsActiveStatus = true).		
Trigger:	The user requests to suspend a vendor profile line-item row by clicking its specific context "Deactivate" trigger button asset.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user requests to suspend a vendor profile line-item row by clicking its		

	specific row item context "Deactivate" trigger button.		
		Step 2: The frontend interrupts the execution flow and displays a confirmation warning alert overlay modal presenting the text: "Deactivate vendor [Vendor Name]? It will no longer appear when creating projects."	
	Step 3: The user clicks the "Deactivate" confirmation action trigger control within the modal pop-up window block framework. [ALT]		**Step 4:** The client-side interface captures the click event. The Angular VendorService builds and dispatches an asynchronous HTTP PATCH request to the backend service controller path api/Vendor/deactivate/{id} passing the entity's primary key identifier. The API endpoint routes processing workflows to the data layer. The runtime references the master ApplicationDbContext instance to establish database connections and search-targets the

			matching vendor entity records index row.
			Step 5: The backend server runs internal logic validation rules. The application checks that the targeted vendor record exists in the database table and evaluates active operational data associations to ensure the vendor profile isn't currently assigned to an ongoing project workflow.
			Step 6: If all validation rules clear successfully, the system updates the targeted entity tracking field values, changing the status flag column to false (IsActiveStatus = false), and commits the state transaction via an asynchronous database command (SaveChangesAsync).
			Step 7: The API shifts execution pipelines and delivers an HTTP 200 OK network success response payload. The client application catches the response, updates the local directory layout caches, hides the modal overlay window component from the DOM view, and pops a green success toast notification stating: "Vendor record

			successfully flagged deactivated."
Alternate courses:	**[ALT] Step 3:** The user aborts the suspension and clicks the "Cancel" button control asset inside the confirmation dialog window block layout. The modal container layer closes immediately, the workflow halts, and the vendor's active database status remains completely unchanged.		
Conclusion:	The use case concludes once all data integrity and dependency validations clear, the target tracking column updates successfully, and local frontend directory caches reload.		
Post-condition:	• The specific vendor record status column is permanently flagged as false (IsActiveStatus = false), completely rendering the entry hidden from all future transactional selection picklists.		
Business rules:	• The system strictly blocks deactivation requests targeting any vendor profile with active downstream project ties to prevent relational database mapping breaks.		
Implementation constraints and specifications:	• Relational lookup constraints must evaluate active project arrays via Entity Framework Core tracking contexts ahead of committing state alterations.		
Assumptions:	• None		
Open issues:	• None		

Table 57: Deactivate Vendor - Technical Use Case Narrative

2.6.7 Create Vendor Type

INTEGRATED NGO MANAGEMENT SYSTEM		
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**
Use case name:	Create Vendor Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
Use case id:	6.7	
Priority:	High	
Source:	EPI-USE	
Primary business actor (PBA):	• Super Admin	
Primary system actor (PSA):	• None	
Other participating actors:	• None	
Other interested stakeholders:	Admin	
Description:	This use case details the system flow where an authorized user establishes a new category classification label within the vendor lookup directories. The user populates an open Angular Reactive Form Group interface layout detailing alphanumeric controls for Classification Type Name, Functional Description Mapping Scope, and Initial Status. Upon clicking submit, the client interface checks form control validation syntax states and dispatches an asynchronous HTTP POST request to the backend API endpoint (POST api/VendorType/create) packing a structured CreateVendorTypeRequest JSON payload body. The .NET 8 backend controller handles the arriving payload, performs unique naming verification checks against the database using the ApplicationDbContext, maps parameters into a new VendorType data model row instance, and executes an asynchronous save update command. The use case concludes when the server returns an HTTP 201 Created code, local frontend array caches update, form states clear, and a green confirmation toast notification fires.	
Pre-condition:	• The executing operator must possess valid administrative system configuration privileges, and session authentication tokens must be valid.	
Trigger:	The user requests to add a new classification category by clicking the "Create Vendor Type" or "Manage Types" action button trigger component asset.	
Typical Course of Events:	**Actor Action**	**System Response** **Manual Action** / **Automated Action**

	Step 1: The user requests to add a new category by clicking the "Create Vendor Type" or "Manage Types" action button trigger component.		
		Step 2: The system launches the type of generation modal wrapper overlay. The frontend generates and renders an open Angular Reactive Form Group interface detailing an alphanumeric text input field for the Classification Type Name, a text area for the Functional Description Mapping Scope, an option selector for Status, a "Discard" button, and a primary "Commit Records" submit button.	

	Step 3: The user populates the classification type name input field, enters descriptive details about the category scope, selects an initial status configuration, and clicks the primary "Commit Records" button. [ALT]		**Step 4:** The frontend framework evaluates active client field validation syntax rules. The Angular VendorService builds and dispatches an asynchronous HTTP POST request to the backend API api/VendorType/create route packing a structured CreateVendorTypeRequest JSON payload body. The backend controller intercepts the incoming payload. The application establishes connections to the storage layer via the ApplicationDbContext interface and triggers an asynchronous read check query against the "VendorType" entity rows to verify that the classification name string does not conflict with an existing entry.
			Step 5: The server backend processes validation checks against data model constraints. The system checks that the category name string is populated and unique, verifies that field lengths do not exceed maximum database character allocations, and confirms the active session token contains necessary administrative privileges.

			Step 6: Upon completing verification loops, the system maps the validated payload parameters into a new instance entry of the VendorType database model table collection and saves the tracking metrics via an asynchronous commit statement (SaveChangesAsync).
			Step 7: The API endpoint returns an HTTP 201 Created code success response. The frontend catches the response message, appends the newly returned vendor type category instance into its local cache data arrays, clears the active modal form state layer, and triggers a green confirmation toast statement reading: "New vendor type category successfully added to global system directories."
Alternate courses:	**[ALT] Step 3:** The user cancels the category generation and clicks the "Discard" button control asset inside the form panel modal drawer. The frontend interface stops outbound payload packaging routines, releases active form variables from memory buffers, prevents network data calls to the server host, and closes the modal view layer.		
Conclusion:	The use case concludes when data models clear validation parameters, the classification record registers successfully inside database backend tables, and configuration lookup dashboards refresh.		
Post-condition:	• A new vendor classification type entity row is permanently registered to the global lookup directories, making it immediately available for sorting vendor profiles.		

Business rules:	• The system blocks the generation of any new vendor type entity containing a category label string that matches an existing entry exactly (case-insensitive) within the table partitions.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 58: Create Vendor Type - Technical Use Case Narrative

2.6.8 Update Vendor Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Update Vendor Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.8		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process for modifying an existing vendor type classification category's name string or description scope metrics. The user adjusts field metrics within an active edit workflow modal overlay window container, populated automatically via an explicit Angular form patchValue operation. Upon clicking save, the client interface checks browser-layer validation rules and dispatches a structured HTTP PUT call directly to the endpoint URL parameter path (PUT api/VendorType/update/{id}). The API endpoint binds to the shared ApplicationDbContext instance, runs an asynchronous lookup transaction seeking a matching primary key sequence, evaluates payload modifications against backend validation constraints, and overwrites the tracked database entity schema properties with the new payload values. The use case concludes when changes commit to disk via an asynchronous update command, the modal popup container closes, and a confirmation alert toast fire.		
Pre-condition:	• The targeted vendor type record instance must exist within the database layer partitions, and the operator must hold valid system configuration management write claims.		
Trigger:	The user requests to alter a vendor category type name or description by clicking the row "Edit" button layout element asset next to a specific listing line item.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

	Step 1: The user requests to alter a vendor category type name or description by clicking the row "Edit" button layout element.		
		Step 2: The frontend triggers the edit workflow modal overlay window container. The system passes the active row parameters and patches the existing database field metrics directly into the form text inputs via an explicit Angular form patch operation (patchValue).	
	Step 3: The user modifies the alphanumeric text input values (updating category names or adjusting functional scope explanations) and clicks the "Save Changes" execution control button. [ALT]		**Step 4:** The client application verifies form control validity on the browser layer. The Angular VendorService dispatches a structured HTTP PUT call directly to the backend endpoint URL parameter path api/VendorType/update/{id}, passing the updated text parameter payload maps. The API endpoint processes data extraction streams. The

			tracking logic binds to the shared ApplicationDbContext instance and runs an asynchronous lookup transaction statement (SingleOrDefaultAsync) seeking a matching vendor type primary key sequence mapping.
			Step 5: The system evaluates payload modifications against backend validation constraints. It checks that the targeted vendor type record is found in the database layer, assures alphanumeric character arrays do not break size properties, and validates that the active session holds necessary configuration management permissions.
			Step 6: Upon completing verification loops, the system overwrites the tracked database entity schema properties with the new payload values and saves changes to the database storage layer via an asynchronous update command (SaveChangesAsync).
			Step 7: The server backend dispatches an HTTP 200 OK status packet back to the client interface. The frontend application intercepts the success code message, closes the modal popup block

			container, updates its local state arrays, and triggers a confirmation alert toast reading "Changes committed successfully."
Alternate courses:	**[ALT] Step 3:** The user aborts mutations and clicks the "Discard" or "Cancel" execution control button. The frontend interface closes the editing panel layout immediately, drops all form changes from memory buffers, and blocks outbound network data transactions to the server host.		
Conclusion:	The use case concludes when the system successfully validates modified parameters, saves updates to the database storage layer, closes active modal boxes, and refreshes the main dashboard lookup grid views.		
Post-condition:	• The target vendor type category's database entity fields are permanently updated with the new input text parameters, modifying classification text across downstream dashboard views.		
Business rules:	• Configuration mutations targeting global classification directories are restricted strictly to users with active administrative configuration or system management privileges.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 59: Update Vendor Types - Technical Use Case Narrative

2.6.9 View Vendor Types

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	View Vendor Types	**USE CASE TYPE**	
Use case id:	6.9	Business Requirements: □	
Priority:	High	System Analysis: □	
Source:	EPI-USE	System Design: ✓	
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case describes how the system extracts and lists all registered vendor type classification profiles from backend database layers to render them on dashboard views or fill selection picklists. Upon loading the target lookup view or a vendor form requiring classification picklists, the client framework initializes initialization lifecycle code hooks (ngOnInit). This triggers the internal Angular service layer to build and dispatch an asynchronous HTTP GET transaction to the backend server routing endpoint (GET api/VendorType/list). The .NET 8 backend controller handles the request, references the master ApplicationDbContext, runs an optimized select query statement applying sorting criteria, bundles matching rows into an array data transfer block, and dispatches an HTTP 200 OK status code back to the client thread. The use case finishes once the frontend loops through returned structures via *ngFor row generation commands, rendering rows into an interactive table layout grid.		
Pre-condition:	• The user must be authenticated within the application and maintain active network data connectivity streams linking host server environments.		
Trigger:	The user opens the vendor type category configuration dashboard view, navigating to the lookup panel or loading an administrative input form requiring category arrays.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

	Step 1: The user opens the vendor type category configuration dashboard view (either by navigating to the lookup panel or loading a vendor creation form that requires category picklists).		
			Step 2: The Angular engine workspace shell loads the component framework layout and triggers automated initialization lifecycle code blocks (ngOnInit).
			Step 3: The internal Angular VendorService constructs and dispatches an asynchronous HTTP GET transaction to the backend server routing endpoint api/VendorType/list. The backend controller routes processing workflows to data layers. The system references the master ApplicationDbContext and constructs a query expression evaluating active rows across the global VendorType data sets.
			Step 4: The database management system runs an optimized select query statement using

			Entity Framework Core, processing sorting parameters, and assembling list items into a clean dataset transfer block.
			Step 5: The server bundles the matching rows into an array collection dataset transfer block and returns an HTTP 200 OK response to the client.
			Step 6: The Angular template loops through the returned collection using structured *ngFor row generation commands, rendering rows into an interactive table layout grid that displays category names, description text, status badges, and operation management action buttons.
Alternate courses:	None		
Conclusion:	The use case concludes cleanly when serialized collection records compile successfully from databases, updating structural dashboard layouts or picking arrays.		
Post-condition:	• Registered vendor category classification profiles are extracted from backend systems and mapped onto structural interface layouts for query navigation.		
Business rules:	• The collection return payload must include inactive tracking rows when requested inside administrative lookup views but filter out inactive instances when accessed from secondary client picklist creation windows.		
Implementation constraints and specifications:	• None		

Assumptions:	• None
Open issues:	• None

Table 60: View Vendor Types - Technical Use Case Narrative

2.6.10 Reactivate Vendor Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Reactivate Vendor Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.10		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the process invoked when an authorized operator restores a deactivated vendor type classification category back to system-wide visibility. The user requests reactivation from a target row item layout context. The frontend displays a verification prompt overlay block dialog framework. Upon user confirmation, the client application issues an asynchronous HTTP PATCH update request directly to the backend endpoint URL parameter path (PATCH api/VendorType/reactivate/{id}). The .NET 8 backend controller intercepts the transaction command, sets up connections to the storage layer via the ApplicationDbContext, and evaluates internal backend logic validations. Upon verification approval, the system updates the target entity's status tracking column to true (IsActiveStatus = true) and commits updates to persistent relational disk files. The use case finishes when the server dispatches an HTTP 200 OK status code, the client refreshes category displays, hides active modals, and fires a green success toast notification.		
Pre-condition:	• The targeted vendor classification category record must exist in the database with its operational visibility flag currently set to false.		
Trigger:	The user requests to restore a deactivated vendor category by clicking its row "Reactivate" button trigger layout asset component.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

	Step 1: The user requests to restore a deactivated vendor category by clicking its row "Reactivate" button trigger layout asset.		
		Step 2: The frontend framework displays a modal verification prompt overlay showing the text: "Are you sure you want to reactivate vendor type [Category Name]? It will immediately reappear when classifying project vendors."	
	Step 3: The user assesses the prompt context and clicks on the "Confirm Reactivate" action control button inside the confirmation dialog panel overlay. [ALT]		**Step 4:** The front end interface captures the execution choice. The Angular VendorService sends an asynchronous HTTP PATCH update request directly to the backend endpoint URL parameter path api/VendorType/reactivate/{id}. The backend controller handles the arriving transaction request. The application establishes connections to the database storage layer via the ApplicationDbContext

			configuration interface and locates the targeted vendor type record item index row.
			Step 5: The system evaluates internal backend logic validations, verifying that the specific vendor type row exists in the table partitions and assuring the executing user possesses valid configuration workspace privileges.
			Step 6: Upon verification approval, the system updates the target entity tracking field values, toggling the explicit schema state column status indicator flag over to true (IsActiveStatus = true), and commits the state transaction via an asynchronous update command (SaveChangesAsync).
			Step 7: The API sends back an HTTP 200 OK status response. The client application catches the code response message, refreshes the category data display lists, removes the modal view layer from the active layout, and renders a green success toast saying, "Vendor type category reactivated successfully."
Alternate courses:	**[ALT] Step 3:** The user dismisses the reactivation prompt and clicks the "Cancel" or "Discard" button control asset inside the confirmation dialog panel layout overlay. The system immediately closes the modal prompt container, terminates the operational intent, and leaves the target database record parameters completely unchanged.		

Conclusion:	The use case concludes once all systemic validations clear successfully, the classification visibility flag updates within database storage files, prompt frames close, and configuration lookup dashboards refresh.
Post-condition:	• The specific category row tracking index visibility property is permanently flipped to true, making the classification option immediately selectable within vendor profile setup sheets.
Business rules:	• Only users possessing valid configuration management workspace privileges and explicit system modification security claims can re-elevate classification entity status values.
Implementation constraints and specifications:	• State adjustments must execute over an asynchronous pipeline leveraging Entity Framework tracking context blocks (SaveChangesAsync).
Assumptions:	• None
Open issues:	• None

Table 61: Reactivate Vendor Type - Technical Use Case Narrative

2.6.11 Deactivate Vendor Type

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 12 June 2026 **Version: 1.0**	
Use case name:	Deactivate Vendor Type	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	6.11		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case details the step-by-step process executed when an authorized user suspends a vendor classification category type entity row from systemic use, preventing it from classifying future vendor profiles while preserving existing data relationships. The user triggers deactivation from a specific category row item. The frontend generates a confirmation warning alert overlay modal. Upon affirmation, the Angular client interface builds and dispatches an asynchronous HTTP PATCH request to the backend service controller routing path (PATCH api/VendorType/deactivate/{id}). The backend controller intercepts the request and uses the ApplicationDbContext to verify entity status parameters and run dependency evaluation rules. If all rules pass, the context tracker updates the status flag column to false. The use case finishes once table rows update on disk, the modal overlay window closes from the DOM view, and a green success notification statement flashes.		
Pre-condition:	• The targeted vendor type record row must exist inside relational tables with its structural status flag currently matching true (IsActiveStatus = true).		
Trigger:	The user requests to suspend a vendor category by clicking its specific row item context "Deactivate" trigger button layout element asset.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**

	Step 1: The user requests to suspend a vendor category by clicking its specific row item context "Deactivate" trigger button.		
		Step 2: The frontend halts direct executions, generating a confirmation warning alert overlay displaying the message text: "Deactivate vendor type [Category Name]? It will no longer appear when creating or editing vendors."	
	Step 3: The user clicks the "Deactivate" confirmation action trigger control within the confirmation modal pop-up window block framework. [ALT]		**Step 4:** The frontend interface processes the user action event. The Angular VendorService issues an asynchronous HTTP PATCH request to the backend service controller path api/VendorType/deactivate/{id} carrying the primary key parameter. The API endpoint intercept routes processing workflows to the data layer. The master ApplicationDbContext connects to the relational tables and targets the matching

			vendor type entity record index line row.
			Step 5: The system evaluates business rule validations before proceeding. The application executes structural checks to verify that the specific vendor type record row is found and assesses that no active vendor profiles are currently classified under this category to protect downstream system data mapping integrity.
			Step 6: If all validation rules pass, the system updates the record's status flag column to false (IsActiveStatus = false) using the Entity Framework context tracker and commits the state transaction via an asynchronous update command (SaveChangesAsync).
			Step 7: The API sends an HTTP 200 OK status response code. The client interface captures the success message, reloads category directory layouts, hides the modal overlay window, and displays a success notification statement: "Vendor type category successfully flagged deactivated."
Alternate courses:	**[ALT] Step 3:** The operator chooses to cancel the action and clicks the "Dismiss Deactivation" or "Cancel" control within the confirmation modal pop-up window block framework. The overlay layout closes immediately, execution stops, and the category's database fields remain unchanged.		
Conclusion:	The use case concludes once all database validation checks clear, the target row record flag transitions successfully within system storage files, and directory views refresh local state caches.		

Post-condition:	• The target vendor classification row status indicator is permanently written as false (IsActiveStatus = false), removing it from picklist selection filters across systemic vendor creation screens.
Business rules:	• Deactivation requests targeting any vendor type category are strictly blocked by the system if active downstream vendor database profiles depend on the classification index.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 62 :Deactivate Vendor Type - Technical Use Case Narrative

2.7 Project Management Subsystem Use Case Narratives

2.7.1 Create Project

INTEGRATED NGO MANAGEMENT SYSTEM		
AUTHOR:	Alexander Steven	**Date:** 28 April 2025 **Version:** 1.0
USE CASE NAME:	Create Project	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	7.1	
PRIORITY:	High	
SOURCE:	EPI-USE	
Primary Business Actor (PBA):	System Admin	
Primary System Actor (PSA):	• None	
Other Participating Actors:	• None	
Other Interested Stakeholders:	Users in the project	
Description:	This use case describes the administrative process of initializing a new project on the system. The user chooses a project name and description, a start date, and optionally an end date. They then select a project type from a pre-defined list populated via the system API (“GET api/Project Types”) and select a project photo. The process then moves onto the locations step where the geographic route is sequenced. Following this, the user navigates to the teams section where they select from a list of user roles using the system API (“GET api/ProjectRoles”) and subsequently add users to those roles by fetching the user directory via the API (“GET api/User”). The user then defines the project deadlines before concluding with the indemnity forms section. Here, the system retrieves available forms from the system list (“GET api/IndemnityForms”) allowing the admin to attach a mandatory legal waiver. The use case ends when the frontend dispatches the full project hierarchy to the backend via the API (“POST api/Projects”).	
Pre-condition:	• The Admin must be authenticated. Selectable Project Types, Catalog Roles, and Users must exist in the database.	
Trigger:	The Admin selects the "+ Create Project" button on the Project Portfolio dashboard.	
	Actor Action	**System Response**

		Manual Action	Automated Action
Typical Course of Events:	**Step 1:** The System Admin clicks the "+ Create Project" button on the Portfolio dashboard.		**Step 2:** The system loads and displays the Project Identity step using the Angular frontend ("ProjectWizardComponent") with the following controls: • A Project Identity title. • A Project Name label with a text input field and placeholder text "e.g. Rhino Relocation 2026". • A Project Type label with a dropdown selection populated from the API ("GET api/ProjectTypes"). • A Project Profile Picture label with a selectable gallery of 8 project icons. • A Project Start Date label with a date picker. • An Include project end date checkbox which, when selected, displays an End Date date picker. • A Project Visibility Status section with radio buttons for Active Project and Inactive/Hidden.
	Step 3: The Admin enters the project details and clicks the Next Section button.		**Step 4**: The system loads and displays the Team Allocation step using the Angular

			frontend with the following controls: A Team Allocation title. • An Add Project Role button which opens an inline selector for roles retrieved via the API (“GET api/ProjectRoles”). • A list of role blocks, each containing o A role icon and name. o A Create Chat checkbox for enabling SignalR messaging groups. o A list of member chips with a remove “×” option. o An Add Member button which opens an inline dropdown of users retrieved via the API (“GET api/User”).
	Step 5: The Admin enters the project details and clicks the Next Section button.		**Step 6:** The system loads and displays the Project Route step using the Angular frontend with the following controls: • A Project Route title. • An Add Location button. • A sequenced list of location rows, each containing a stop number, an address text input field, and a Delete button.

	Step 7: The Admin adds the sequences the route for the project and clicks the Next Section button.		**Step 8:** The system loads and displays the Project Deadlines step using the Angular frontend with the following controls: • A Project Deadlines title. • An Add Deadline button. • A grid of deadline cards, each containing a name input field, a datetime-local picker, a description textarea, and a remove "x" button.
	Step 9: The Admin defines the deadlines and clicks the Next Section button.		**Step 10:** The system loads and displays the Legal & Compliance step using the Angular frontend with the following controls: • A Legal & Compliance title. • An Indemnity Form label with a dropdown selection of active forms retrieved via the API ("GET api/IndemnityForms"). • A Create Project button.
	Step 11: The Admin clicks the Create Project button. [ALT]		**Step 12**: The system sets the loading state to true and dispatches the project object to the backend API ("POST api/Projects").

			Step 13: The backend ("ProjectsController") receives the payload and iterates through the team roles to set the "Entity State" of pre-existing users and roles to "Unchanged" within the database context ("AppDbContext").
			Step 14: The system adds the new record to the database ("Project" Entity), saves the changes, and returns an HTTP 201 Created response. [ALT]
			Step 15: The system redirects the Admin back to the Portfolio dashboard and displays a success notification.
Alternate courses:	**[ALT] Step 11:** The system detects that the Project Name or Project Type fields are empty. The Create Project button remains disabled, and validation messages are displayed under the respective fields.		
	[ALT] Step 14: The system fails to save the project due to a database connection error. The system displays an error message: "Failed to create project. Check console for details."		
Conclusion:	A new project is successfully created and stored in the system.		
Post-condition:	The project is visible in the Portfolio and available for management in the Command Centre.		
Business Rules:	• Secure Access		
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.		

Assumptions:	• None
Open issues:	• None

Table 63: Create Project - Technical Use Case Narrative

2.7.2 View Project

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Alexander Steven</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Project</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">7.2</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">System Users</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the process where a user selects a specific project from the portfolio list which triggers a request to the backend API (“GET api/Projects/{id}”). The system retrieves all associated data including the project type, assigned locations, chronological deadlines, and the team structure. This information is then displayed in the command centre where the user can monitor the operational status and see the specific roles and members involved. The User can filter by active projects at the top of the screen.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The project must exist in the database.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The User clicks on a project card or selects “View Project” from the Portfolio or from Teams page.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The User clicks the View Project button on a project card.</td><td></td><td>Step 2: The system loads the View project task using the Angular frontend (“ProjectDetailComponent”) and extracts the ID from the route</td></tr>
</table>

			parameters.
			Step 3: The system invokes the project service ("ProjectService") to fetch project details via the API ("GET api/Projects/{id}").
			Step 4: The backend ("ProjectsController") performs an eager-load query on the database ("Project" Entity) using ".Include" statements to gather related locations, deadlines, and team members. [ALT]
			Step 5: The system returns the populated project object to the frontend.
			Step 6: The system renders the Command Center UI with the following panels: • An Identity Panel displaying the project photo, name, type, and start/end dates. • A Team & Personnel Panel showing each role and the assigned members. • A Route Sequence Panel displaying a vertical timeline of addresses.

			• A Critical Deadlines Panel listing milestones by date and time. • A Compliance Panel showing the status of the attached indemnity form. • A status indicator showing a pulse animation for "Active" or "Completed".
Alternate courses:	**[ALT] Step 4:** The system cannot find the project with the specified ID. The API returns an HTTP 404 response, and the frontend redirects the user back to the projects list.		
Conclusion:	The user successfully views the detailed project status.		
Post-condition:	The user is presented with options to edit or close.		
Business Rules:	• Secure Access		
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.		
Assumptions:	• None		
Open issues:	• None		

Table 64: View Project - Technical Use Case Narrative

2.7.3 Update Project

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Alexander Steven		**Date:** 28 April 2025 **Version:** 1.0
USE CASE NAME:	Update Project		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	7.3		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	System Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	Users in the project		
Description:	This use case describes the process of modifying an existing project record. The user chooses to modify an existing project which opens the wizard interface in edit mode. The system populates the form using data retrieved from the backend. The user can update the project identity, change the team allocation by interacting with the project role API ("GET api/ProjectRoles"), or adjust the timeline and locations. Once the changes are finalized, the system updates the record by sending the data to the backend API ("PUT api/Projects/{id}").		
Pre-condition:	• The user must be an authorized System Admin, and the project must already exist.		
Trigger:	The User selects the "Edit Project" button within the View project panel.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The User selects the "Edit Project" button within the View project panel.		**Step 2:** The system navigates to the project wizard in edit mode ("ProjectWizardComponent") and sets "isEditMode" to true.

			Step 3: The system fetches the existing project details via the API ("GET api/Projects/{id}") and populates all wizard controls.
	Step 4: The User modifies the necessary fields across the five wizard steps (Identity, Team, Route, Deadlines, or Legal).		
	Step 5: The User clicks the Update Project button on the final step.		**Step 6:** The system dispatches the updated project object to the backend API ("PUT api/Projects/{id}"). [ALT]
			Step 7: The backend ("ProjectsController") marks the "Project" Entity as modified in the database context ("AppDbContext") and executes "SaveChangesAsync()".
			Step 8: Upon receiving an HTTP 204 No Content response, the system redirects the User back to the Project Portfolio**.**
Alternate courses:	**[ALT] Step 6:** The system fails to validate the updated dates (e.g., end date is before start date). An error message is displayed, and the update is blocked.		
Conclusion:	The project information is successfully updated in the database.		
Post-condition:	The updated details are immediately reflected in the View Project Info page and Projects List page.		

Business Rules:	• Secure Access
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.
Assumptions:	• None
Open issues:	• None

Table 65: Update Project - Technical Use Case Narrative

2.7.4 Deactivate Project

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Alexander Steven</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Project</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">7.4</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">System Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">Users in the project</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the process of suspending a project's operational status. The user selects the toggle option within the Project Info page which triggers a request to the backend API (“PATCH api/Projects/{id}/toggle-active”). The system modifies the “IsActive” Boolean on the “Project” Entity in the database and saves the change. Once the update is confirmed, the frontend updates the status indicator and the pulse animation to reflect that the project is now “Completed”.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be registered on the system, and their account must be active.
• The User must be a System Admin
• The project must exist and currently be active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">A user wishes to deactivate a project, so they click on “Deactivate”</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The User clicks the Deactivate Project button in the Project Info page.</td><td></td><td>Step 2: The System asks the Admin for confirmation they want to deactivate the project.</td></tr>
<tr><td>Step 3: The Admin confirms or clicks back</td><td></td><td>Step 4: The system invokes the toggle</td></tr>
</table>

			method in the project detail component ("ProjectDetailComponent") which calls the project service.
			Step 5: The system dispatches a request to the backend API ("PATCH api/Projects/{id}/toggle-active").
			Step 6: The backend ("ProjectsController") retrieves the project from the database ("Project" Entity) and inverts the "IsActive" status.
			Step 7: The system executes "SaveChangesAsync()" on the database context ("AppDbContext") and returns the new status in an HTTP 200 OK response.
			Step 8: The system updates the frontend UI to change the button text to "Activate Project".
			Step 9: The system updates the status pulse indicator from a green "Online" state to a grey "Off grid" state.
Alternate courses:	**[ALT] Step 3**: The Admin clicks back and returns to the Project View Page		
	[ALT] Step 6:		

	The system cannot find the project record to update. The API returns an HTTP 404 response and the status remains unchanged.
Conclusion:	The project is successfully moved to an inactive state.
Post-condition:	The project remains in the database but is marked as inactive in the portfolio.
Business Rules:	• Secure Access
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.
Assumptions:	• None
Open issues:	• None

Table 66: Deactivate Project - Technical Use Case Narrative

2.7.5 Create Project Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Alexander Steven</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Project Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">7.5</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">System Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the creation of a new project category used to classify different NGO initiatives. The user enters a unique name and a detailed description for the project type. Upon saving, the frontend dispatches the data to the system API (“POST api/ProjectTypes”). The backend receives the request, adds the new category to the “ProjectType” Entity, and commits the record to the SQL database. This new type then becomes available for selection when administrators are using the project creation wizard.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin must be registered on the system, and their account must be active.
• The Admin must be authenticated.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects the “Create Project Type” option in the Admin Dashboard</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The Admin clicks the Create Project Type button on the View Project Types page</td><td></td><td>Step 2: The system loads and displays the Create Project Type modal using the Angular frontend with the following controls:</td></tr>
</table>

			• A Create Project Type title. • A Name label with a text input field. • A Description label with a textarea field. • A Save button.
	Step 3: The User enters the name (e.g., "Wildlife Relocation") and the description and clicks save.		**Step 4**: The system dispatches the project type object to the backend API ("POST api/ProjectTypes").
			Step 5: The backend ("ProjectTypesController") adds the new record to the database ("ProjectType" Entity) via the database context ("AppDbContext"). [ALT]
			Step 6: The system executes "SaveChangesAsync()" and returns the created object with an HTTP 200 OK response.
			Step 7: The system closes the modal and refreshes the project type list to include the new entry.
Alternate courses:	**[ALT] Step 5:** The system detects that a project type with the same name already exists in the database. The API returns an error, and the user is prompted to choose a unique name.		
Conclusion:	A new project type is added to the system catalog.		

Post-condition:	The project type can now be assigned to new projects.
Business Rules:	• Secure Access
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.
Assumptions:	• None
Open issues:	• None

Table 67: Create Project Type - Technical Use Case Narrative

2.7.6 View Project Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Alexander Steven</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Project Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">7.6</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">System Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the administrator views the list of project categories currently configured in the system. The user navigates to the project type management page, which triggers a request to the backend API (“GET api/ProjectTypes”). The system retrieves all records from the “ProjectType” Entity and presents them in a grid view, showing the name and description of each type.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• Project types must exist in the database.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin navigates to the Project Types management section.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The Admin selects Project Types from the management menu or Admin dashboard.</td><td></td><td>Step 2: The system loads the project type list component and extracts the data from the API (“GET api/ProjectTypes”).</td></tr>
<tr><td></td><td></td><td>Step 3: The backend (“ProjectTypesController”) executes a SQL Read Query to fetch all records from the</td></tr>
</table>

			database (“ProjectType” Entity). [ALT]
			Step 4: The system returns a collection of project types to the frontend.
			Step 5: The system renders the Project Type Management screen with the following controls: • A Project Types title. • A Search bar to filter types by name. • A list of project type cards or rows, each displaying the Name and Description. • Action buttons for Edit and Delete on each row. • An Add New Type button.
Alternate courses:	**[ALT] Step 3:** No project types are found in the database. The system displays an empty state message: “No project types found. Create a new mission category to get started.”		
Conclusion:	The user successfully views the configured project categories.		
Post-condition:	The user can proceed to update or delete specific project types.		
Business Rules:	• Secure Access		
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.		
Assumptions:	• None		
Open issues:	• None		

Table 68: View Project Type - Technical Use Case Narrative

2.7.7 Update Project Type

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Alexander Steven		**Date:** 28 April 2025 **Version:** 1.0
USE CASE NAME:	Update Project Type		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	7.7		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	System Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes the process of modifying the details of an existing project category. The user chooses to modify a project type from the management list which triggers a call to the system API (“GET api/ProjectTypes/{id}”). The user updates the name and description through the Angular frontend modal. Once the user saves the changes, the system dispatches the updated object to the backend using the .NET 8.0 framework via the API (“PUT api/ProjectTypes/{id}”). The backend updates the “EntityState” of the “ProjectType” Entity to modified within the database context (“AppDbContext”) and persists the changes to the SQL database.		
Pre-condition:	• The project type must exist in the database.		
Trigger:	The Admin selects the “Edit” option next to a project type in the management list.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Step 1: The Admin clicks the Edit button on a specific project type row.		**Step 2:** The system loads the Update Project Type modal using the Angular frontend and populates the controls with

			existing data retrieved from the API ("GET api/ProjectTypes/{id}") .
			Step 3: The system displays the following controls within the modal: • An Update Project Type title. • A Name label with a text input field containing the current name. • A Description label with a textarea field containing the current description. • A Save Changes button. • A Cancel button.
	Step 4: The User modifies the name or description fields and clicks the Save Changes button. [ALT]		**Step 5:** The system dispatches the updated data to the backend API ("PUT api/ProjectTypes/{id}") built on the .NET 8.0 framework.
			Step 6: The backend ("ProjectTypesController") receives the payload and marks the "ProjectType" Entity as modified in the database context ("AppDbContext").
			Step 7: The system executes

			"SaveChangesAsync()" and returns an HTTP 204 No Content response upon success. [ALT]
			Step 8: The system closes the modal and refreshes the management list to display the updated information.
Alternate courses:	**[ALT] Step 4:** The user clicks the Cancel button. The system closes the modal without saving any changes and the record remains in its original state.		
	[ALT] Step 7: The system fails to save the changes due to a database concurrency conflict. An error message is displayed: "The record you attempted to edit was modified by another user."		
Conclusion:	The project type details are successfully updated.		
Post-condition:	The updated project type name and description are visible in the management list and the project wizard.		
Business Rules:	• Secure Access		
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.		
Assumptions:	• None		
Open issues:	• None		

Table 69: Update Project Type - Technical Use Case Narrative

2.7.8 Deactivate Project Type

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Alexander Steven</td><td>Date: 28 April 2025
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Project Type</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">7.8</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">System Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">Users in the project</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the administrative removal of a project category from the system. The user chooses to remove a project type from the list which triggers a request to the system API (“DELETE api/ProjectTypes/{id}”) using the .NET 8.0 framework. The backend identifies the specific record in the “ProjectType” Entity and removes it from the database context (“AppDbContext”). Once the change is committed to the SQL database, the project type is no longer available for selection during project creation.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The project type must exist and should not be currently assigned to any active projects to avoid referential integrity errors.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects the “Delete” or “Deactivate” option for a project type in the management list.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The Admin clicks the Delete button next to a project type entry.</td><td></td><td>Step 2: The system displays a confirmation modal using the Angular frontend with the following controls:
• A Confirm Deletion title.</td></tr>
</table>

			• A message asking if the user is sure they want to remove the project type. • A Delete button. • A Cancel button.
	Step 3: The Admin clicks the Delete button. [ALT]		**Step 4**: The system dispatches a request to the backend API ("DELETE api/ProjectTypes/{id}") developed with the .NET 8.0 framework.
			Step 5: The backend ("ProjectTypesController") locates the entry in the "ProjectType" Entity via the database context ("AppDbContext").
			Step 6: The system removes the record from the context and executes "SaveChangesAsync()" to update the SQL database. [ALT]
			Step 7: Upon a successful deletion, the API returns an HTTP 204 No Content response.
			Step 8: The system closes the modal and removes the entry from the management list.

Alternate courses:	**[ALT] Step 3:** The user clicks the Cancel button. The confirmation modal closes and no data is sent to the API.
	[ALT] Step 6: The system prevents deletion because the project type is linked to existing project records. An error message is displayed: "Cannot delete project type as it is currently in use by one or more projects."
Conclusion:	The project type is successfully removed from the system.
Post-condition:	The project type no longer appears in the management list or the project creation wizard or on the project type list.
Business Rules:	• Secure Access
Implementation constraints and specifications:	• JWT tokens must be securely processed and handled purely in memory and local storage on the client.
Assumptions:	• None
Open issues:	• None

Table 70: Deactivate Project Type - Technical Use Case Narrative

2.7.9 Add Indemnity Form

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 22 April 2026 **Version: 1.0**	
Use case name:	Add Indemnity Form	**USE CASE TYPE** Business Requirements: □ System Analysis: □ ✓ System Design: ✓	
Use case id:	7.9		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case allows the Super Admin to add a new indemnity form to the system by uploading a PDF document and capturing its related details. The Angular frontend presents a reactive form allowing the capture of metadata and a PDF file upload. Upon submission, the system transmits a multipart/form-data request to the ASP.NET Core API endpoint. The backend processes the file using the IFormFile interface, validates the file type, and persists the file path alongside metadata in the IndemnityForm entity via Entity Framework Core. The system ensures uniqueness constraints and enforces business rules before committing the transaction to the SQL Server database. The form becomes part of the project documentation structure and can be linked to a project during the capturing indemnity process.		
Pre-condition:	• The Super Admin must be logged into the system. • The Super Admin must have permission to manage indemnity forms. • The indemnity form must be available as a valid PDF document. • The form name must not already exist in the system.		
Trigger:	The Super Admin selects the “Add Indemnity Form” option.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin navigates		**Step 2:** The Angular Router loads the

INTEGRATED NGO MANAGEMENT SYSTEM			
	to the indemnity management interface via the Angular frontend.		IndemnityFormComponent. The system renders a reactive form containing: • Form Name input field • Description input field • File upload control (PDF only) • Submit button
	Step 3: The Super Admin enters the form name, description, and uploads a PDF file, then selects submit. [ALT]		**Step 4:** The Angular service constructs a multipart/form-data payload and sends a POST request to: POST /api/IndemnityForms
			Step 5: The API controller receives the request. The system performs validation: • Confirms file is not null • Validates file extension (pdf) Validates form name uniqueness via EF Core query. [ALT]
			Step 6: The system processes the uploaded file: • Generates a unique filename using GUID • Stores the file in the server directory (e.g. /indemnity-forms/) Construct a file path string. [ALT]

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 7: The system creates a new IndemnityForm entity: • IndemnityFormId (PK) • FormName • Description • FilePath • CreatedAt • IsActive
			Step 8: The system persists the entity.
			Step 9: The API returns HTTP 200 OK with the created object.
			Step 10: The frontend displays a success toast notification.
Alternate courses:	**[ALT] Step 3:** The Super Admin uploads a non-PDF file. The system rejects the file and returns HTTP 400 Bad Request with message: "Only PDF files are allowed."		
	[ALT] Step 5: A duplicate form name is detected. The system aborts the process and returns HTTP 400 with message: "An indemnity form with this name already exists."		
	[ALT] Step 6: File storage fails. The system returns HTTP 500 Internal Server Error, and no database record is created.		
Conclusion:	The indemnity form is successfully uploaded and made available for project use.		
Post-condition:	• A new indemnity form record exists in the system. • The uploaded PDF is stored and linked to the indemnity form record. • The form status is active.		
Business rules:	• Only a Super Admin may add indemnity forms. • Only PDF documents may be uploaded. • Each indemnity form name must be unique. • Newly added forms must be active by default.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Implementation constraints and specifications:	• The system must support secure PDF upload. • The file path must be stored in the database.
Assumptions:	• The serve has available storage space. • The project documentation structure has already been defined.
Open issues:	• Should uploaded forms require approval before becoming active?

Table 71: Add Indemnity Form - Technical Use Case Narrative

2.7.10 Update Indemnity Form

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Amahle Khoza</td><td colspan="2">Date: 22 April 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Update Indemnity Form</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □ ✓
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.10</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">• Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Administrator</td></tr>
<tr><td>Description:</td><td colspan="3">This use case allows the Super Admin to update an existing indemnity form’s detail or replace an PDF document. This is necessary when the wording, legal requirements, project conditions, or document details change. The update ensures that the system continues to use the most accurate and relevant version of the indemnity form.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Super Admin must be logged into the system.
• The indemnity form must already exist,
• The Super Admin must have permission to update indemnity forms.
• If replacing then document, the new file must be a PDF.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Super Admin selects an existing indemnity form and chooses “Update Indemnity Form.”</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The Super Admin navigates to the indemnity form list and selects</td><td></td><td>Step 2: The Angular Router loads the UpdateIndemnityFormComponent. The system retrieves the selected form via:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	a form to update.		GET /api/IndemnityForms/{id}
			Step 3: The system populates a reactive form with: • Form Name • Description • Existing file reference Optional file upload control
	Step 4: The Super Admin modifies the form details and optionally uploads a new PDF file. [ALT]		**Step 5:** The Angular service constructs a multipart/form-data payload and sends: PUT /api/IndemnityForms/{id}
			Step 6: The API controller retrieves the existing record using EF Core. [ALT]
			Step 7: The system validates: • Form name uniqueness • File format (if provided) • Required fields [ALT]
			Step 8: If a new file is uploaded: • Generate GUID filename • Replace stored file path • Save file to server directory [ALT]
			Step 9: The system updates the entity fields and executes.
			Step 10: The API returns HTTP 200 OK.
			Step 11: The frontend displays a success notification.
Alternate courses:	**[ALT] Step 4:** Invalid file format uploaded. System returns HTTP 400 Bad Request (“Only PDF files allowed”)		
	[ALT] Step 6: Indemnity form not found. System returns HTTP 404 Not Found		

INTEGRATED NGO MANAGEMENT SYSTEM	
	[ALT] Step 7: Duplicate form name. System returns HTTP 400 Bad Request
	[ALT] Step 8: File upload failure. System returns HTTP 500 Internal Server Error
Conclusion:	The indemnity form is successfully uploaded and made available for project use.
Post-condition:	• A new indemnity form record exists in the system. • The uploaded PDF is stored and linked to the indemnity form record. • The form status is active.
Business rules:	• Only a Super Admin may update indemnity forms. • Form names must remain unique. • Only PDF replacement files are allowed. • A form linked to active projects should not be replaced without confirmation.
Implementation constraints and specifications:	• The system must validate all updated fields before saving. • Database changes and file replacement should be handled carefully to avoid inconsistency.
Assumptions:	• The replacement PDF has been reviewed and approved. • The system has access to the existing stored file.
Open issues:	• Should users be notified when an indemnity form changes?

Table 72: Update Indemnity Form - Technical Use Case Narrative

2.7.11 View Indemnity Form

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 22 April 2026 **Version: 1.0**	
Use case name:	View Indemnity Form	**USE CASE TYPE**	
Use case id:	7.11	Business Requirements: □	
Priority:	High	System Analysis: □ ✓	
Source:	EPI-USE	System Design: ✓	
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case allows authorised users to view indemnity forms stored in the system. The Super Admin may view all forms for management purposes, while other authorised users may view active forms relevant to their projects.		
Pre-condition:	• The user must be logged into the system. • At least one indemnity form must exist. • The user must have permission to view indemnity forms. • The uploaded PDF file must still be available in storage.		
Trigger:	The user selects “View Indemnity Form” or opens the indemnity form list.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user navigates to the indemnity form section.		**Step 2:** Angular component loads and triggers: GET /api/IndemnityForms [ALT]
			Step 3: API executes EF Core query.
			Step 4: The system returns a list of forms.

INTEGRATED NGO MANAGEMENT SYSTEM			
	Step 6: The user selects a form to view.		**Step 7:** The system retrieves file path and opens PDF. [ALT]
			Step 8: Angular component loads and triggers: GET /api/IndemnityForms [ALT]
Alternate courses:	**[ALT] Step 2:** No forms exist. System returns empty list		
	[ALT] Step 7: File missing. System returns error message		
Conclusion:	The user successfully views the indemnity form details and, where available, the uploaded PDF.		
Post-condition:	• No data is changed. • The selected indemnity form remains available in the system. • The user has accessed the relevant document.		
Business rules:	• Users may only view forms they are authorised to access. • Inactive forms should not be visible to general users. • Super Admins may view both active and inactive forms for management purposes. • A form must have a valid file path before it can be opened.		
Implementation constraints and specifications:	• The system must prevent unauthorised direct access to restricted documents. • The frontend must support viewing or downloading PDF files.		
Assumptions:	• The document was uploaded successfully. • The user has a stable connection to load the file. • Permissions are correctly configured.		
Open issues:	• Should the system record when a user views a form? • Should inactive forms be hidden completely or only marked as inactive?		

Table 73: View Indemnity Form - Technical Use Case Narrative

2.7.12 Remove Indemnity Form

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 22 April 2026 **Version: 1.0**	
Use case name:	Remove Indemnity Form	**USE CASE TYPE** Business Requirements: □ System Analysis: □ ✓ System Design: ✓	
Use case id:	7.12		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case allows the Super Admin to remove an indemnity form from active use. Instead of permanently deleting the document, the system should preferably deactivate it so that historical project records remain intact. This protects data integrity, especially where the indemnity form may already be linked to existing projects.		
Pre-condition:	• The Super Admin must be logged into the system. • The indemnity form must exist. • The Super Admin must have permission to remove indemnity forms. • The system must check whether the form is linked to any projects.		
Trigger:	The Super Admin selects “Remove Indemnity Form.”		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The Super Admin selects a form.		**Step 2:** System retrieves form via EF Core. [ALT]
			Step 3: System checks for project links
	Step 4: The Super Admin confirms removal. [ALT]		**Step 5:** System updates: form.IsActive = false;

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 6: System saves changes.
			Step 7: System returns HTTP 200 OK.
			Step 8: System retrieves form via EF Core. [ALT]
Alternate courses:	**[ALT] Step 2:** Form not found. HTTP 404		
	[ALT] Step 4: User cancels. Process terminates		
Conclusion:	The indemnity form is removed from active use without compromising historical records.		
Post-condition:	• The indemnity form status is inactive. • The form can no longer be attached to new projects.		
Business rules:	• Only a Super Admin may remove indemnity forms. • Forms linked to projects should not be permanently deleted. • Inactive forms must not be available for new captures.		
Implementation constraints and specifications:	• The user interface should clearly distinguish inactive forms. • Soft delete should be implemented using an active or inactive status field. • The system must check dependencies before removal.		
Assumptions:	• Inactive records are excluded from normal user selection lists.		
Open issues:	• Should inactive forms be restorable?		

Table 74: Remove Indemnity Form - Technical Use Case Narrative

2.7.13 Capture Indemnity Form -

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Amahle Khoza	**Date:** 22 April 2026 **Version: 1.0**	
Use case name:	Capture Indemnity Form	**USE CASE TYPE** Business Requirements: □	
Use case id:	7.13	System Analysis: □ ✓	
Priority:	High	System Design: ✓	
Source:	EPI-USE		
Primary business actor (PBA):	• Super Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case allows the Super Admin to capture an indemnity form against a specific project by linking an existing active indemnity form to that project. This ensures that each project can be associated with the correct legal or consent documentation required for participation, compliance, or project activities. Unlike adding a form, which uploads the document, capturing a form means assigning or attaching an existing form to a project.		
Pre-condition:	• The Super Admin must be logged into the system. • The Super Admin must have permission to manage project documentation. • The project must exist and be active. • At least one active indemnity form must exist. • The selected indemnity form must not already be linked to the selected project.		
Trigger:	The Super Admin selects “Capture Indemnity Form” from a project.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Super Admin selects a project.		**Step 2:** System retrieves project details. [ALT]
			Step 3: System retrieves active forms. [ALT]

INTEGRATED NGO MANAGEMENT SYSTEM			
			Step 4: System displays selectable forms.
			Step 5: Super Admin selects form and confirms.
			Step 6: System validates: • Project exists • Form exists • No duplicate link [ALT]
			Step 7: System creates: ProjectIndemnityForm: • ProjectId • IndemnityFormId • CapturedAt
			Step 8: System saves via EF Core.
			Step 9: Returns HTTP 200 OK.
Alternate courses:	**[ALT] Step 2:** Project not found. HTTP 404		
	[ALT] Step 3: No forms available. Prompt to add form		
	[ALT] Step 6: Duplicate capture. HTTP 400		
Conclusion:	The selected indemnity form is successfully captured against the project and becomes part of that project's documentation.		
Post-condition:	• A ProjectIndemnityForm record exists. • The project is linked to the selected indemnity form. • The captured form can be viewed as part of the project documentation.		
Business rules:	• Only a Super Admin may capture indemnity forms for projects. • Only active indemnity forms may be captured. • Only active projects may be linked to indemnity forms.		

INTEGRATED NGO MANAGEMENT SYSTEM	
	• A project cannot have the same indemnity form captured more than once. • Capture does not create a new form; it links an existing form to a project.
Implementation constraints and specifications:	• A linking table should be used to connect Project and IndemnityForm. • A unique constraint should prevent duplicate project-form combinations. • The system must validate both ProjectId and IndemnityFormId before saving.
Assumptions:	• Projects are already created before forms are captured. • The correct indemnity form has already been uploaded.
Open issues:	• Should a project be allowed to have multiple indemnity forms? • Should users link to the project be notified when a form is captured?

Table 75: Remove Indemnity Form - Technical Use Case Narrative

2.7.14 Capture Project Feedback -

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Alexander Steven	**Date:** 15 July 2026 **Version:** 1.0	
Use case name:	Capture Project Feedback	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	7.14		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	System Admin (Authenticated)		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	A System Admin can attach a project feedback form to a project which will make it so that anyone who has been added to that project will be sent a form 1 week after the project commences asking them for a rating and comments. The system captures the selection for all involved staff. The System Admin navigates to the Feedback screen, sees the list of projects, and toggles the "Enable feedback" checkbox. The Angular frontend instantly sends a PUT request to the .NET 8 API to update the project's feedback configuration in the database.		
Pre-condition:	• The System Admin is authenticated and navigating the Feedback screen. The selected project must exist and be active.		
Trigger:	The System Admin clicks the "Enable feedback" checkbox next to a project name.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The System Admin clicks the "Feedback" block on the dashboard.		**Step 2:** The system loads the Project Feedback screen, retrieving and displaying the projects and indicating which

			projects currently have feedback enabled. [ALT]
	Step 3: The System Admin clicks the "Enable feedback" checkbox on the project they wish to add feedback to.		**Step 4:** The Angular "FeedbackService" captures the toggle event and dispatches a "PUT api/project/{id}/feedback-status" request to the backend with the Boolean payload "isEnabled": true.
			Step 5: The API controller receives the request, validates the System Admin's JWT claims, and invokes Entity Framework Core to locate the target Project entity. [ALT]
			Step 6: The system updates the "FeedbackEnabled" Boolean property on the entity to true and calls "SaveChangesAsync()".
			Step 7: The API returns an HTTP 204 No Content status. The frontend registers the success, leaves the checkbox in the checked state, and displays a success toast notification.
Alternate courses:	**[ALT] Step 2:** The system fails to retrieve the projects from the database. The system returns an HTTP 500 Internal Server Error and displays an empty list with a red error toast: "Failed to load projects."		
	[ALT] Step 5a: The project ID cannot be found in the database. The system returns an HTTP 404 Not Found response. The frontend reverts the checkbox to its previous state and displays a red error toast: "Project not found."		
	[ALT] Step 5b:		

	The API returns an HTTP 500 Internal Server Error due to a database connection failure while updating. The frontend reverts the checkbox and displays: "System error, please try again."
Conclusion:	The project is successfully flagged to collect feedback, ensuring participants receive the form when the project commences.
Post-condition:	The “FeedbackEnabled” property on the Project entity is set to true. Users in the project receive the review after the project commences
Business rules:	Only System Admins can enable project feedback or view the results of the feedback. All comments are anonymous.
Implementation constraints and specifications:	• The API endpoint only needs to receive a simple true or false value (a Boolean) to update the status, instead of sending all the project's details over the network. This keeps the system fast and simple.
Assumptions:	• We assume that the project already has staff members assigned to it, and that their email addresses are correct on the system so they can receive the feedback form.
Open issues:	• The system does not currently save a record in the "AuditLog" table when a System Admin turns the feedback checkbox on or off.

Table 76: Capture Project Feedback - Technical Use Case Narrative

2.7.15 View Project Feedback -

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Project Feedback</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.15</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• Project Staff</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">If a project has the "FeedbackEnabled" setting turned on, the system automatically sends a feedback survey to all involved users one week after the project starts, or if a System Admin ends the project early. This use case allows a System Admin to view the results of these surveys. The Admin clicks the "Feedback" widget on the home page, which loads a list of projects. When a specific project is clicked, the system retrieves the "ProjectFeedback" records from the database, calculates the average star rating, and displays the rating along with all anonymous comments.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in. The project has "FeedbackEnabled" turned on, and users have submitted feedback.
• The Project has ended, either early by the admin, or by one week and triggered by the time delay built into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Feedback" widget on the home dashboard to view the results of feedback from projects.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
</table>

	Step 1: The System Admin clicks the "Feedback" widget on the home page.		**Step 2:** The system loads the Project Feedback screen, showing a list of projects on the left side of the screen.
	Step 3: The System Admin clicks on a specific project from the list.		**Step 4:** The Angular frontend asks the database for the feedback data by sending a GET request to api/feedback/project/{id}. [ALT]
			Step 5: The .NET 8 API checks that the user is a System Admin and then looks up all the "ProjectFeedback" records for that specific project. [ALT]
			Step 6: The system calculates the average star rating from all the retrieved records and bundles the anonymous comments together to send back.
			Step 7: The API sends back an HTTP 200 OK success message along with the data.
			Step 8: The frontend updates the screen to show a large average star rating (e.g., "4.5 Based on 10 reviews") and a list of the anonymous comments on the right side of the screen.
			Step 12: In the current implementation the “Save
Alternate courses:	**[ALT] Step 4:** The system has trouble fetching the data, we will just display a “Feedback currently unavailable right now” message with a gif to still look professional and to let the user know they aren’t making a mistake.		
	[ALT] Step 5: The database connection fails or the project cannot be found. The API returns an HTTP 500 Internal Server Error or HTTP 404 Not Found. The		

	system leaves the results area empty and shows a red error message: "Failed to load project feedback."
Conclusion:	The System Admin can clearly read the average rating and anonymous comments to see how the project is performing.
Post-condition:	The feedback data is displayed on the screen. No information in the database is changed.
Business rules:	Only System Admins can view the project feedback results.
Implementation constraints and specifications:	• To protect user privacy, the API must never send user IDs or names back with the feedback comments.
Assumptions:	• The system correctly counts how many reviews were submitted and calculates the average accurately.
Open issues:	• If a project has thousands of comments, loading them all at once might slow down the page because there are currently no "next page" buttons (pagination).

Table 77: View Project Feedback - Technical Use Case Narrative

2.7.16 Create Impact Feedback Form

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Create Impact Feedback Form</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.16</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Community Partners</td></tr>
<tr><td>Description:</td><td colspan="3">A System Admin can create a custom impact form to gather data from community partners. The Admin selects to create a "Questionnaire" (for surveys) or an "Attendance Board" (for tracking daily status). The system provides a form builder where the Admin can continuously add and edit questions (for a questionnaire) or attendees and custom statuses (for an attendance board). The Admin then configures a sending schedule (e.g., "Once" or "Weekdays" with an end date) and saves the form. The system stores the layout as a "FormStructureJson" string in the database and automatically extracts and saves the initial student roster if it is an attendance board.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in and navigating the Impact Forms screen.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Blank form" or a template button on the dashboard to create a new impact form.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The System Admin clicks "Blank form" and chooses either</td><td></td><td>Step 2: The Angular frontend loads the form builder page matching the chosen type.</td></tr>
</table>

	"Questionnaire" or "Attendance Board" from the pop-up modal.		
	Step 3: The System Admin enters the form title, description, receiver contact details, and the sending schedule (e.g., Frequency: Weekdays, Send Time: 07:00).		
	Step 4: The System Admin uses the builder to add content. They can loop this step to add multiple questions and options (for Questionnaires) or add multiple attendees and custom statuses (for Attendance Boards).		**Step 5:** The Angular frontend temporarily stores these form blocks and schedule settings in memory.
	Step 6: The System Admin clicks the "Save & Publish" button. [ALT]		**Step 7:** The frontend packages the data into an ImpactFormDto object and sends a POST api/ImpactForms request to the .NET 8 backend.
			Step 8: The API controller receives the request, and saves the ImpactForm record in the database, storing the layout inside the "FormStructureJson" field. [ALT]
			Step 9: The system reads the saved JSON. If the form type is

			"attendance", the system loops through the JSON to find the initial attendees and saves each one into the FormStudents database table so their attendance can be tracked.
			Step 10: The API returns an HTTP 200 OK response. The frontend takes the user back to the main forms list and displays a success toast message.
Alternate courses:	**[ALT] Step 6:** The System Admin leaves a required field blank (like the form title). The system highlights the missing field in red and prevents the save.		
	[ALT] Step 8: The database fails to save the record due to a server error. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast: "Failed to save form."		
	[ALT] Step 10: We display an error message		
Conclusion:	The custom impact form is successfully built, saved, and ready to be filled out by the receiver based on the schedule.		
Post-condition:	A new ImpactForm record is saved in the database. If it is an attendance board, FormStudent records are also created.		
Business rules:	Only System Admins can create and publish impact forms.		
Implementation constraints and specifications:	• The form's visual layout and questions are stored as a single JSON string ("FormStructureJson") to keep the database flexible, rather than creating separate database tables for every question and option.		
Assumptions:	• The receiver's email address is valid so they can receive the form link.		
Open issues:	• There is currently no way to save a "Draft"; clicking save immediately publishes the form to the schedule. • No way to verify the receiver's email yet		

Table 78: Create Impact Feedback Form - Technical Use Case Narrative

2.7.17 View Impact Feedback Forms

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Impact Forms</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.17</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A System Admin can view all the past projects and see the impact forms questions and answers and see what project they are assigned to. The system retrieves records for impact data and displays it neatly for the admin. The Admin clicks the "Impact Forms" widget on the dashboard, which triggers the Angular frontend to request all available forms from the .NET 8 API. The backend calculates how many responses each form has received. The system then displays the forms as cards, allowing the Admin to click and view the structure, questions, and assigned project details of any specific form.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in and is on the main Dashboard. Impact forms must already exist in the database.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Impact Forms" widget on the home dashboard.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The System Admin clicks the "Impact Forms" widget on the home page.</td><td></td><td>Step 2: The Angular frontend sends a GET request to api/ImpactForms to fetch the forms.</td></tr>
</table>

			Step 3: The API controller receives the request, verifies the Admin's access, and queries the ImpactForms table in the database using Entity Framework Core. [ALT]
			Step 4: For each form, the system checks if it is an attendance board or a questionnaire. It then calculates the "ResponsesCount" by checking either the StudentAttendances or ImpactFormResponses tables.
			Step 5: The API packages this data into an ImpactFormDto list and returns an HTTP 200 OK success response. [ALT]
			Step 6: The Angular frontend receives the data and neatly displays the folders and "Recent forms" as cards. Each card shows the form's name, the number of responses, and the project it is linked to.
	Step 7: The System Admin clicks on a specific form card to view its internal questions and structure.		**Step 8:** The frontend parses the "FormStructureJson" string for that form and visually renders the questions and available

			options for the Admin to read.
Alternate courses:	**[ALT] Step 3:** The database is down or an unexpected server error occurs. The API returns an HTTP 500 Internal Server Error. The Angular frontend catches the error, displays an empty dashboard, and shows a red error toast message to the Admin: "Failed to load impact forms."		
	[ALT] Step 5: The query succeeds but no forms exist yet. The API returns an empty list [] with an HTTP 200 OK. The frontend displays a friendly message to the Admin: "No forms available. Create a new one to get started."		
Conclusion:	The System Admin can successfully view all impact forms, see how many responses they have, and inspect the specific questions asked on each form.		
Post-condition:	The impact forms are displayed on the screen. No database records are modified.		
Business rules:	Only System Admins can view the impact forms dashboard and structure.		
Implementation constraints and specifications:	• The system calculates the "ResponsesCount" directly in the database query to ensure the page loads quickly without pulling down unnecessary response data.		
Assumptions:	• The frontend correctly interprets the "FormStructureJson" to display the questions exactly as the community partner will see them.		
Open issues:	• If there are hundreds of forms, the system currently fetches all of them at once, which may slow down the initial page load.		

Table 79: View Impact Feedback Forms - Technical Use Case Narrative

2.7.18 Update Impact Feedback Forms

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Update Impact Feedback Form</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.18</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Community Partners</td></tr>
<tr><td>Description:</td><td colspan="3">The System Admin can update an existing impact form that can be added to projects to make form completions and capturing easier. The system shows fields for the form name and the questions currently on the form as editable content. The Admin opens the form builder, which is pre-filled with the current form details. The Admin can loop through and edit, add, or remove questions, or change the sending schedule. Once saved, the Angular frontend sends a PUT request to the .NET 8 API to overwrite the old form details and "FormStructureJson" in the database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in and viewing the impact forms list. The form being updated must already exist in the database.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Edit" button on a specific impact form card.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The System Admin clicks "Edit" on a specific impact form.</td><td></td><td>Step 2: The Angular frontend opens the form builder and pre-fills it with the existing form's data (Title,</td></tr>
</table>

			Schedule, and Questions).
	Step 3: The System Admin edits the form. They can loop this step to add new questions, delete old ones, or change the schedule.		**Step 4:** The Angular frontend temporarily holds these changes in memory, updating the local "FormStructureJson".
	Step 5: The System Admin clicks the "Save & Publish" button. [ALT]		**Step 6:** The frontend checks that no required fields are blank, packages the updated data into an ImpactFormDto, and sends a PUT api/ImpactForms/{id} request to the backend.
			Step 7: The .NET API controller receives the request and asks Entity Framework Core to find the existing form in the database. [ALT]
			Step 8: The system overwrites the old details with the new data and calls SaveChangesAsync(). [ALT]
			Step 9: The API returns an HTTP 204 No Content success response.
			Step 10: The frontend takes the user back to the dashboard and displays a green success toast message:

			"Form updated successfully."
Alternate courses:	**[ALT] Step 5:** The System Admin clears out a required field (like the form title) and clicks save. The frontend stops the save process and highlights the missing field in red.		
	[ALT] Step 7: The form ID cannot be found in the database (e.g., someone else deleted it). The API returns an HTTP 404 Not Found response. The frontend shows a red error toast: "Form not found."		
	[ALT] Step 8: The database fails to save the updates due to a connection issue. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast: "Failed to update form."		
Conclusion:	The custom impact form is successfully updated with the new questions and scheduling rules.		
Post-condition:	The existing ImpactForm record in the database is modified.		
Business rules:	Only System Admins can edit existing impact forms.		
Implementation constraints and specifications:	• Modifying the "FormStructureJson" is allowed, but the system must be careful not to break how old responses are displayed if a question is completely deleted.		
Assumptions:	• Changing the schedule rules on an existing form will only apply to future send dates, not past ones.		
Open issues:	• If a question is deleted, it is unclear if the old answers to that question will still be visible in the results dashboard.		

Table 80: Update Impact Feedback Form - Technical Use Case Narrative

2.7.19 Deactivate Impact Feedback Forms

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Deactivate Impact Feedback Form</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.19</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• System</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">The System Admin can deactivate an impact form from the list of all the forms being displayed; this can remove unused forms and clear clutter from the system. The system loads the form name for deactivation confirmation before removing it from view. When confirmed, the Angular frontend sends an HTTP DELETE request to the .NET 8 API. The backend uses Entity Framework Core to locate the record in the "ImpactForms" table, permanently removes it, and saves the changes. The system then alerts the Admin that the form was successfully removed.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in and viewing the list of recent impact forms. The form must currently exist in the database.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Deactivate" (or delete) button on a specific impact form card.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The System Admin clicks the "Deactivate" button on a specific impact form.</td><td></td><td>Step 2: The Angular frontend pauses the action and displays a confirmation modal asking: "Are you sure</td></tr>
</table>

			you want to deactivate this form?"
	Step 3: The System Admin clicks the "Confirm" button. [ALT]		**Step 4:** The Angular frontend sends a DELETE api/ImpactForms/{id} request to the backend.
			Step 5: The API controller receives the request, verifies the Admin's permissions, and uses Entity Framework Core's FindAsync to locate the target record in the "ImpactForms" table. [ALT]
			Step 6: The system executes the Remove() command on the record and calls SaveChangesAsync() to commit the deletion to the database. [ALT]
			Step 7: The API returns an HTTP 204 No Content success response.
			Step 8: The Angular frontend receives the success response, immediately removes the form's card from the screen, and displays a green success toast: "Form successfully deactivated."

Alternate courses:	**[ALT] Step 3:** The System Admin changes their mind and clicks "Cancel". The confirmation modal closes, no request is sent, and the form remains on the screen.
	[ALT] Step 5: The form ID cannot be found in the database. The API returns an HTTP 404 Not Found response. The frontend closes the modal and displays a red error toast: "Form not found. It may have already been removed."
	[ALT] Step 6: The database fails to delete the form (for example, if there are locked database constraints). The API returns an HTTP 500 Internal Server Error. The frontend leaves the form on the screen and displays a red error toast: "Failed to deactivate form. Please try again."
Conclusion:	The selected impact form is completely removed from the system and vanishes from the Admin's view.
Post-condition:	The specific ImpactForm record is deleted from the database.
Business rules:	Only System Admins have the permission to deactivate/delete impact forms.
Implementation constraints and specifications:	• The backend strictly uses the DELETE HTTP method to perform the removal.
Assumptions:	• The frontend automatically updates the list without needing to refresh the entire webpage.
Open issues:	• If a form has already collected responses, deleting it might cause data issues if the database doesn't have "Cascade Delete" turned on for the response tables.

Table 81: Deactivate Impact Feedback Form - Technical Use Case Narrative

2.7.20 Capture Impact Feedback

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Capture Impact Feedback</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.20</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Community Partner</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">System Administrator</td></tr>
<tr><td>Description:</td><td colspan="3">The system will capture an email address or phone number to send the form to the community partner. When the scheduled time arrives, the community partner receives a unique public link to the form. Clicking the link opens the Angular frontend, which asks the .NET 8 API to retrieve the specific form using its "PublicLinkGuid". The system displays either a Questionnaire or an Attendance Board. The partner fills in the answers or marks attendance statuses (like Bike, Walk, Absent) and clicks submit. The system sends the data back to the API to be saved in the database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The impact form has been created, published, and the community partner has received their unique public link.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Community Partner clicks the public link in their email and submits the form.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The Community Partner clicks the public link provided in their email.</td><td></td><td>Step 2: The Angular frontend opens and sends a GET api/ImpactForms/public/{guid} request to the backend.</td></tr>
</table>

			Step 3: The API controller looks up the "ImpactForm" in the database using the GUID, retrieves the "FormStructureJson", and returns it to the frontend. [ALT]
			Step 4: The system reads the JSON and displays the correct layout on the screen (either survey questions or an attendance roster).
	Step 5: The Community Partner answers the questions or marks the daily attendance statuses, then clicks "Submit".		**Step 6:** The Angular frontend packages the answers. If it is a questionnaire, it creates an "ImpactFormResponseDto". If it is an attendance board, it creates a list of "SyncAttendanceDto" records.
			Step 7: The frontend sends a POST request to either the submit endpoint or the attendance/sync endpoint on the .NET API.
			Step 8: The API receives the data and saves the new records into either the "ImpactFormResponses" or "StudentAttendances" database tables using Entity Framework Core. [ALT]
			Step 9: In The API returns an HTTP 200 OK success message and The frontend hides the form and displays a green success message: "Response submitted successfully. Thank you!"
Alternate courses:	**[ALT] Step 3:**		

	The public link is invalid or the form was deactivated. The API returns an HTTP 404 Not Found error. The frontend displays a red error message: "This form is no longer available or the link is incorrect."
	[ALT] Step 8: The database fails to save the answers due to a connection error. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast: "Failed to submit response. Please try again."
Conclusion:	The community partner successfully captures and submits their impact data to the NGO's database.
Post-condition:	New data is stored in the "ImpactFormResponses" or "StudentAttendances" tables.
Business rules:	Anyone with the exact public link can submit the form without needing to log in (no authentication required).
Implementation constraints and specifications:	• Questionnaires are saved as raw JSON strings in the responses table, while Attendance records are saved relationally with specific dates and statuses.
Assumptions:	• The community partner has a stable internet connection to load and submit the form.
Open issues:	• Because the public link does not require a login, there is a risk of the link being shared and filled out by unauthorized people

Table 82: Capture Impact Feedback- Technical Use Case Narrative

2.7.21 View Impact Feedback

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Impact Feedback</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.21</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A System Admin can view all of the project's impact results and see the impact form questions and answers, the assigned project, and which community partner answered. The system retrieves and displays these records neatly. The Admin clicks the "Results" button on a specific form card on the dashboard. Depending on whether the form is a questionnaire or an attendance board, the Angular frontend requests the appropriate data from the .NET 8 API. The system fetches the data and dynamically renders visual charts, graphs, and breakdowns on the Results Dashboard.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The System Admin is logged in and viewing the impact forms dashboard.
• The form must have collected responses or attendance data.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The System Admin clicks the "Results" button on a specific impact form card.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The System Admin clicks the "Results" button</td><td></td><td>Step 2: The Angular frontend opens the Results Dashboard and sends a GET request to the backend. It calls</td></tr>
</table>

	on an impact form card.		api/ImpactForms/{id}/responses for surveys, or api/ImpactForms/{id}/attendance-data for attendance boards.
			Step 3: The API controller receives the request, verifies the Admin's JWT token, and queries either the "ImpactFormResponses" or "StudentAttendances" tables in the database using Entity Framework Core. [ALT]
			Step 4: The system gathers the records and returns them in an HTTP 200 OK success response.
			Step 5: The Angular frontend receives the data and uses it to draw the charts. Questionnaires show option breakdowns (like pie charts), while Attendance Boards show a Daily Register graph and Individual Attendee horizontal bar charts. [ALT]
Alternate courses:	**[ALT] Step 3:** The database query fails due to a server connection issue. The API returns an HTTP 500 Internal Server Error. The frontend displays an empty results pane and shows a red error toast: "Failed to load feedback results."		
	[ALT] Step 5: The query succeeds but no responses have been submitted yet. The API returns an empty list []. The frontend displays an empty state message: "No data available yet. Waiting for responses."		
Conclusion:	The System Admin successfully views the visual charts and breakdowns of the feedback data.		
Post-condition:	The feedback data is displayed on the UI. No records are changed in the database.		
Business rules:	Only System Admins can view the impact feedback results and charts.		
Implementation constraints and specifications:	• The backend uses two different endpoints to keep the data structured properly for the different types of visual charts required.		

Assumptions:	• The frontend charting library can handle large amounts of attendance records to draw the dynamic trend lines correctly.
Open issues:	• If a form has been running for multiple years, loading all the attendance data at once might slow down the rendering of the line graphs.

Table 83: View Impact Feedback - Technical Use Case Narrative

2.7.22 Add Photo

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Add Photo</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.22</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user can upload multiple photos to a project's album. The system loads a page where images can be dragged, dropped, or uploaded from the device. The user clicks the "+ Upload Photos" button while inside a specific album and selects multiple image files from their computer. The Angular frontend packages these files and sends them to the .NET 8 API. The backend loops through the files, assigns a unique GUID to each filename to prevent overwriting, physically saves the images to the server's "wwwroot/uploads/photos" folder, and records the file paths in the "Photos" database table.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is logged in, viewing a specific photo album, and has image files ready to upload.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "+ Upload Photos" button inside a photo album.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "+ Upload Photos" button.</td><td></td><td>Step 2: The system opens the device's native file explorer, allowing the user to select one or multiple image files.</td></tr>
<tr><td>Step 3: The user selects the files and</td><td></td><td>Step 4: The Angular frontend appends the files and the user's ID to a FormData object</td></tr>
</table>

	confirms the upload.		and sends a POST api/PhotoAlbums/{id}/upload request to the backend.
			Step 5: The API controller receives the request and checks if the album exists in the database. [ALT]
			Step 6: The system loops through the uploaded files. For each file, it generates a unique name using a GUID, creates a file stream, and copies the physical file into the server's "wwwroot/uploads/photos" directory.
			Step 7: For each saved file, the system creates a new "Photo" database record containing the album ID, the new "FilePath", and the ID of the user who uploaded it.
			Step 8: The system calls SaveChangesAsync() to commit the new records to the database and the API returns an HTTP 200 OK success message. [ALT]
			Step 9: The Angular frontend refreshes the album view to display the newly uploaded photos and shows a green success toast: "Photos uploaded successfully."
Alternate courses:	**[ALT] Step 5:** The album ID cannot be found (e.g., it was deleted by another user). The API returns an HTTP 404 Not Found response with the message "Album not found." The frontend displays a red error toast.		
	[ALT] Step 8:		

	The server runs out of disk space, or the database connection fails during saving. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast: "Failed to upload photos."
Conclusion:	The selected photos are successfully uploaded, safely stored on the server, and displayed in the project's album.
Post-condition:	Physical image files are saved in the "wwwroot/uploads/photos" folder, and corresponding "Photo" records are added to the database.
Business rules:	Users can only upload photos to albums they have permission to access.
Implementation constraints and specifications:	• Files must be appended to a multipart/form-data request so the .NET API can bind them to the List<IFormFile> parameter correctly.
Assumptions:	• The server has write permissions configured for the "wwwroot/uploads/photos" directory.
Open issues:	• There is currently no restriction on file size or file type (e.g., preventing malicious .exe uploads) explicitly shown in the controller code.

Table 84: Add Photo - Technical Use Case Narrative

2.7.23 View Photo

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Photo</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.23</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user can see all photos from a specific project. A user can click on a project to see all photos from that project; they can then click the project to see all photos in that album. All projects are displayed in a table with their names and a representative project photo. The Angular frontend first requests all "PhotoAlbum" records from the .NET 8 API and displays them as cards with their "CoverPhotoPath". When the user clicks on an album, the system makes a second request to fetch all the individual "Photo" records linked to that album ID, displaying the physical images neatly on the screen.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is logged in. Photo albums and photos must exist in the database and file system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "Photos" tab on the main navigation menu.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "Photos" tab to open the albums page.</td><td></td><td>Step 2: The Angular frontend sends a GET api/PhotoAlbums request to the backend.</td></tr>
<tr><td></td><td></td><td>Step 3: The API controller asks Entity Framework Core to fetch all "PhotoAlbum"</td></tr>
</table>

			records from the database. [ALT]
			Step 4: The system returns an HTTP 200 OK response with the album details, and the frontend displays the albums as interactive cards.
	Step 5: The user clicks on a specific album card (e.g., "Bike drop 2026").		**Step 6:** The frontend sends a second request: GET api/PhotoAlbums/{id}/photos using the selected album's ID.
			Step 7: The API queries the "Photos" database table for all images linked to that specific "PhotoAlbumId". [ALT]
			Step 8: The API returns an HTTP 200 OK response containing a list of the photo file paths.
			Step 9: The Angular frontend renders a neat grid layout displaying all the retrieved images to the user.
Alternate courses:	**[ALT] Step 3:** The database connection fails while loading albums. The API returns an HTTP 500 Internal Server Error. The frontend displays an empty page and a red error toast: "Failed to load albums."		
	[ALT] Step 7: The database connection fails while loading the specific photos inside an album. The API returns an HTTP 500 Internal Server Error. The frontend displays a red error toast: "Failed to load photos for this album."		
Conclusion:	The user successfully navigates the gallery and views the stored photos.		
Post-condition:	The images are displayed on the UI. No records in the database are modified.		
Business rules:	Users can only view photos from albums they have permission to access.		
Implementation constraints and specifications:	• The "FilePath" stored in the database is used by the frontend to fetch the physical images hosted in the "wwwroot/uploads/photos" folder.		

Assumptions:	• The physical image files on the server match the "FilePath" records in the database perfectly (no files were manually deleted from the server folder outside of the system).
Open issues:	• If an album contains hundreds of high-resolution photos, loading them all at once on a single page might be slow for users with weak internet connections.

Table 85: View Photo - Technical Use Case Narrative

2.7.24 Delete Photo

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Delete Photo</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>7.24</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user can remove photos from a project's image folder. A user can click on a project to see all photos and then click the delete photos button; the system then displays an "X" button on photos they uploaded to remove them. When the user confirms the deletion, the Angular frontend sends a DELETE request to the .NET 8 API. The backend retrieves the "Photo" record, permanently deletes the physical image file from the server's "wwwroot/uploads/photos" directory, removes the record from the database, and saves the changes. The system captures the selection and updates the album.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is logged in, has navigated to a project's photo album, and the photo exists on the server.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "X" (Delete) button on a specific photo.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "X" button on a specific photo they wish to remove.</td><td></td><td>Step 2: The Angular frontend pauses the action and displays a confirmation modal asking if they are sure they want to delete the photo.</td></tr>
</table>

	Step 3: The user clicks the "Confirm" button. [ALT]		**Step 4:** The Angular frontend sends a DELETE api/PhotoAlbums/photos/{photoId} request to the backend.
			Step 5: The API controller receives the request and uses Entity Framework Core to find the "Photo" record in the database using the provided ID. [ALT]
			Step 6: The system checks the "FilePath" stored in the database, locates the physical file in the "wwwroot/uploads/photos" folder, and executes System.IO.File.Delete() to permanently remove it from the server.
			Step 7: The system removes the "Photo" record from the database and calls SaveChangesAsync().[ALT]
			Step 8: The API returns an HTTP 204 No Content success response.
			Step 9: The Angular frontend removes the deleted photo from the screen's grid and displays a green success toast: "Photo successfully deleted."
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel" on the modal. The frontend closes the prompt and stops the deletion process.		
	[ALT] Step 5: The photo ID cannot be found in the database. The API returns an HTTP 404 Not Found response. The frontend closes the modal and displays a red error toast: "Photo not found."		
	[ALT] Step 7: The database connection drops while saving the changes. The API returns an HTTP 500 Internal Server Error. The frontend leaves the photo on the screen and displays a red error toast: "Failed to delete photo."		
Conclusion:	The selected photo is permanently deleted from both the server's physical file system and the SQL database.		

Post-condition:	The "Photo" record is destroyed, and the physical file is permanently erased.
Business rules:	The user is deleting photos they uploaded or are a system admin.
Implementation constraints and specifications:	• The physical file must be deleted *before* the database record is removed to ensure no orphaned files are left taking up hard drive space on the server.
Assumptions:	• The backend server has the correct write/delete permissions configured for the "wwwroot/uploads/photos" folder.
Open issues:	• There is currently no "Recycle Bin" or soft delete for photos. • Once the physical file is deleted, it cannot be easily recovered.

Table 86: Delete Photo - Technical Use Case Narrative

2.7.25 Add Photo Album

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 28 July 2026 **Version:** 1.0	
USE CASE NAME:	Add Photo Album	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	7.25		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	Project Team members		
Description:	This use case describes how an authorized user creates a new photo album container within a project. The user enters album details in a modal form, and upon frontend validation and backend processing, the system creates the record, initializes the media storage path, logs the audit entry, and confirms creation.		
Pre-condition:	• The user must be logged into the system and have access to the target active project.		
Trigger:	The user selects the "Add Album" option inside a project's media section.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user navigates to the Project Media gallery tab.		**Step 2:** The system invokes UC 7.26 (View Photo Album) to load existing photo albums associated with the project via an HTTP GET request (PhotoAlbumController – GetAlbumsByProjectId).

	Step 3: The user selects the "Add Album" button.		**Step 4:** The system displays the Add Photo Album modal containing: • Modal title: "Add Photo Album" • Album Name field (Required, text input) • Album Description field (Optional, textarea input) • Action buttons: Create, Cancel. The Create button remains disabled until mandatory fields are filled.
	Step 5: The user enters the album details and selects the "Create" button. [ALT]		**Step 6:** The Angular frontend validates that the Album Name is non-empty and compliant with naming policies. [ALT]
			Step 7: The system routes the request via the Angular PhotoAlbumService to the Web API endpoint (PhotoAlbumController – CreatePhotoAlbum). The backend validates uniqueness within the project and persists the record to SQL Server using Entity Framework Core. **PhotoAlbum Entity:** • PhotoAlbumId (PK) • ProjectId (FK)

			• Name • Description • CreatedByUserId (FK) • CreatedAt
			Step 8: The system writes a log to the AuditLog table capturing the operation user ID, timestamp, and details ("Created Photo Album [Name] for Project [ID]").
			Step 9: The system displays a toaster notification: *"Photo album created successfully."*
Alternate courses:	**[ALT] Step 5:** The user selects "Cancel" or closes the modal. Terminate use case.		
	[ALT] Step 6: Validation fails (e.g., duplicate album name within project). An inline error message is displayed: *"An album with this name already exists in this project."* Go back to Step 4.		
Conclusion:	A new photo album record is created, linked to the project, and ready to receive image uploads.		
Post-condition:	The new album appears in the project's media view.		
Business Rules:	• Only project contributors or admins can create albums. • Album names must be unique per project.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None.		

Table 87: Add Photo Album - Technical Use Case Narrative

2.7.26 View Photo Album

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 28 July 2026 **Version:** 1.0	
USE CASE NAME:	View Photo Album	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	7.26		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes how a user views a project's photo album list or opens a specific album to view contained thumbnail images and metadata.		
Pre-condition:	• The user must be logged into the system and have read access to the project.		
Trigger:	The user selects a project's media/photo section or clicks a specific photo album card.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user selects a project photo album from the gallery grid.		**Step 2:** The system sends an HTTP GET request via the Angular HTTP Client to PhotoAlbumController – GetAlbumDetails with PhotoAlbumId.
			Step 3: The backend verifies user JWT authorization claims and retrieves album metadata along with child photo records

			(.AsNoTracking()) from SQL Server via EF Core. **Entities Involved:** • **PhotoAlbum:** PhotoAlbumId, Name, Description, CreatedAt • **Photo:** PhotoId, PhotoAlbumId (FK), FilePath, UploadedByUserId (FK), UploadedAt [ALT]
			Step 4: The system renders the album viewer page displaying album metadata, cover details, and a grid of photo thumbnails with light-box expansion capabilities.
Alternate courses:	**[ALT] Step 3:** The requested album ID is invalid or deleted. System displays a notification: *"Photo album not found."* Terminate use case.		
Conclusion:	The album contents and associated image media are retrieved and rendered on screen.		
Post-condition:	The user views album contents without changing underlying data.		
Business Rules:	• Secure Access (User must have read permissions for the project).		
Implementation constraints and specifications:	• Lazy loading or thumbnail optimization for fast image gallery load times.		
Assumptions:	• None		
Open issues:	• None.		

Table 88: View Photo Album - Technical Use Case Narrative

2.7.27 Update Photo Album

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 28 July 2026 **Version:** 1.0	
USE CASE NAME:	Update Photo Album	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	7.27		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes how a user modifies existing photo album details (e.g., album title or description). The changes are validated, updated in the database, and logged in audit history.		
Pre-condition:	• The user is logged in, has access to the project.		
Trigger:	The user clicks the "Edit Album" button on an existing photo album card/header.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user clicks the "Edit Album" action button on an album card.		**Step 2:** The system displays the Edit Photo Album modal pre-populated with existing data (Name and Description fields).
	Step 3: The user modifies the fields and clicks the "Save Changes" button. [ALT]		**Step 4:** The system validates that the mandatory Name field is non-empty. [ALT]
			Step 5: The system sends an HTTP PUT

			request (PhotoAlbumController – UpdatePhotoAlbum). The backend validates authorization rules, verifies that the user is the creator or Super Admin, and updates the database record via EF Core.
			Step 6: The system creates an entry in the AuditLog table ("Updated Photo Album [ID] details").
			Step 7: The system closes the modal and displays a toaster notification: *"Photo album updated successfully."*
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel". Changes are discarded. Terminate use case.		
	[ALT] Step 4: Validation fails (e.g., blank title). Error message displayed: *"Album name cannot be empty."* Go back to Step 2.		
Conclusion:	Photo album details are updated in SQL Server and reflected across the UI.		
Post-condition:	The photo album reflects updated metadata.		
Business Rules:	• Admin can update it.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None.		

Table 89: Update Photo Album - Technical Use Case Narrative

2.7.28 Remove Photo Album

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Amahle Khoza	**Date:** 28 July 2026 **Version:** 1.0	
USE CASE NAME:	Remove Photo Album	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	7.28		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Administrator		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes how an authorized user deletes a photo album along with its associated photo file references. The user confirms deletion via a modal, and the system purges the database records and logs the deletion action.		
Pre-condition:	• The user is logged in, has access to the project.		
Trigger:	The user clicks the "Delete Album" button.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user selects the "Delete Album" option on a photo album.		**Step 2:** The system displays a confirmation modal displaying: • Title: "Confirm Delete Album" • Message: *"Are you sure you want to delete '[Album Name]'? All photos in this album will be permanently removed."* • Action buttons: Confirm, Cancel.

	Step 3: The user selects the "Confirm" button. [ALT]		**Step 4:** The request is routed via Angular PhotoAlbumService via an HTTP DELETE call to PhotoAlbumController – DeletePhotoAlbum.
			Step 5: The API verifies permissions, removes the album record and child photo entries from SQL Server via EF Core cascading delete, and triggers cloud/file storage cleanup for physical image files.
			Step 6: The system records an entry in AuditLog ("Deleted Photo Album [ID] and associated media").
			Step 7: The system closes the modal and displays a toaster notification: *"Photo album updated successfully."*
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel". Changes are discarded. Terminate use case.		
Conclusion:	The photo album and its linked photos are deleted from the system.		
Post-condition:	The photo album and its images are permanently removed.		
Business Rules:	• Admin can delete a photo album. • Confirmation modal is mandatory to prevent accidental data loss.		
Implementation constraints and specifications:	• Cascading deletion logic must clean up database records and associated file storage assets cleanly.		
Assumptions:	• None		
Open issues:	• None.		

Table 90: Remove Photo Album - Technical Use Case Narrative

2.8 K9 Subsystem Use Case Narratives

2.8.1 Add Dog

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Add Dog</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.1</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a super admin creates a new K9 within the system. The system allows the user to enter the K9's details, upload an optional image, assign K9 roles, and optionally select the K9's parents. The system then validates the entered information, confirms the creation, and saves the K9 to the database. The K9 is stored along with its associated roles and image, the K9 list is refreshed, and a confirmation message is displayed. The transaction is also recorded in the audit log.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to add a new dog to the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to add a new K9 to the system.</td><td></td><td>Step 2: The system invokes UC 8.3 (View Dogs) to display the K9 page showing all the registered K9s. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API</td></tr>
</table>

			endpoint in Visual Studio (K9Controller – GetAllK9s).
	Step 3: The super admin selects the "+ Add K9" button.		**Step 4:** The system loads and displays the Create K9 modal with the following components: • Create K9 modal title. • Name section: ○ Name label. ○ Name input field. ○ Placeholder text: "Dog Name". • Breed section: ○ Breed label. ○ Breed input field. ○ Placeholder text: "Breed". • Gender section: ○ Gender label. ○ Dropdown list. ▪ Default option: "Select gender". ▪ Male. ▪ Female. • K9 Roles section: ○ K9 Roles label. ○ Multi-select dropdown displaying all available K9 Roles. ○ Selected role counter.

			• Date of Birth section: ○ Date of Birth label. ○ Date selector. • Father section: ○ Father label. ○ Dropdown list. ▪ Default option: "No Father Assigned". ▪ Displays all male K9s retrieved from the system. • Mother section: ○ Mother label. ○ Dropdown list. ▪ Default option: "No Mother Assigned". ▪ Displays all female K9s retrieved from the system. • Image section: ○ Image label. ○ Choose Image button. ○ Displays the selected image file name. • Action buttons: ○ Create button. ○ Cancel button.

			The Father and Mother dropdown lists are populated using the K9 data that was already retrieved when UC 8.3 (View Dogs) was invoked. The K9 Roles dropdown is populated using data retrieved from the K9 Role Entity via the K9 Role Service, which calls the backend API. **K9 Role Entity** • K9RoleId (PK) • RoleName
	Step 5: The super admin enters the K9's details, optionally uploads an image, selects the required K9 roles, and selects the "Create" button. [ALT]		**Step 6:** The system validates the entered details using Angular frontend validation before allowing the K9 to be created. The system validates the following: • The K9 Name field is not null or empty. • The K9 Name contains alphabetic characters only. • The Breed field is not null or empty. • The Breed contains alphabetic characters only. • The Date of Birth is not in the future. • If a father is selected, the father must be at least two years older than the K9. • If a mother is selected, the mother must be at least two years older than the K9.

			• If an image is selected, only JPG, JPEG, PNG or WEBP image formats are accepted. If any validation fails, an error message is displayed within the Create K9 modal. [ALT]
			Step 7: The system displays a custom Angular confirmation modal after successful validation. The confirmation modal contains: • Confirmation modal title ("Confirm Create K9"). • Confirmation message label: ○ Displays the message: "Are you sure you want to create this K9?" • Action buttons: ○ Confirm button. ○ Cancel button.
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system captures the new K9's details and constructs a FormData object containing the K9 information, optional image, selected parent K9s and selected K9 Roles. The Angular frontend uses the K9 Service and Angular HTTP Client to route the request to the corresponding HTTP POST API endpoint in Visual Studio (K9Controller – CreateK9). The API endpoint passes the request to the K9 Repository where additional business validation is performed before the data is saved using Entity

			Framework Core in Microsoft SQL Server. The backend validates that: • A K9 cannot be its own parent. • The selected father and mother cannot be the same K9. • The selected father exists and is male. • The selected mother exists and is female. • The selected parents are each at least two years older than the K9. If an image is supplied, the system stores the uploaded image in the application's uploads directory and generates a unique file name. If no image is supplied, a default K9 image is assigned. The system creates a new Animal record together with the corresponding K9 record and stores the selected K9 Roles by creating records in the K9K9Role junction table. The following entities are involved in the process: **Animal Entity** • AnimalId (PK) • AnimalName • AnimalSpecies

			• AnimalGender • AnimalDateOfBirth • AnimalPhoto • ActiveStatus **K9 Entity** • K9Id (PK) • AnimalId (FK) • K9DadId (FK) • K9MomId (FK) **K9K9Role Entity (Junction Table)** • K9Id (PK, FK) • K9RoleId (PK, FK) **K9 Role Entity** • K9RoleId (PK) • RoleName [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the new K9. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a new K9 has been created. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK)

			• Timestamp • LogEntry
			Step 11: The system displays a toaster notification confirming that the K9 has been successfully created. The confirmation message displays the text: "K9 created successfully."
Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the “Cancel” button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and an error message is displayed within the Create K9 modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm and selects the “Cancel” button. Go back to step 4.		
	ALT-Step 9: The system fails to save the new K9 and displays a toaster message: “Failed to create K9”. Terminate use case.		
Conclusion:	The new dog’s details have been saved onto the system, and a confirmation message is shown.		
Post-condition:	The system saves the new K9 in the Animal and K9 table, stores any associated K9 role assignments, uploads and stores the selected image (if provided), and makes the newly created K9 available for viewing and management within the system.		
Business rules:	• Only super admin or users given appropriate permissions can add a new dog.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 91: Add Dog - Technical Use Case Narrative

2.8.2 View Dog

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">View Dog</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.2</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Handler</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a handler views the details of a specific K9 within the system. The system allows the handler to search for or filter the available K9s, select a K9 from the displayed list, and view its details. The system retrieves the selected K9's information from the database and displays it in a View K9 Details modal.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The handler must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The handler wants to view the specific dog’s details on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The handler wants to view a specific K9's details on the system.</td><td></td><td>Step 2: The system invokes UC 8.3 (View Dogs) to display all the current K9s. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetAllK9s). The retrieved K9s are then displayed as K9 cards on the K9 page.</td></tr>
</table>

	Step 3: The handler searches for a K9 by entering its name or breed in the search field or filters the displayed K9s by gender or status. [ALT]		**Step 4:** The system filters and displays the K9s matching the entered search criteria and selected filters. The filtering is performed within the Angular frontend using the retrieved K9 data. [ALT]
	Step 5: The handler selects a K9 card.		**Step 6:** The system retrieves the selected K9's details through the K9 Service in the Angular frontend. The service sends a request via the Angular HTTP Client to the HTTP GET endpoint in Visual Studio (K9Controller – GetK9ById), which retrieves the K9 based on the selected K9Id. The API then retrieves data from the following entity: **Animal Entity:** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalGender • AnimalDateOfBirth • AnimalPhoto • ActiveStatus **K9 Entity:** • K9Id (PK) • AnimalId (FK) • K9DadId (FK)

			• K9MomId (FK) The API also retrieves the K9's assigned roles from the K9K9Role junction table and the related K9Role entity. **K9K9Role Entity (Junction Table):** • K9Id (PK, FK) • K9RoleId (PK, FK) **K9Role Entity:** • K9RoleId (PK) • RoleName The Angular frontend also invokes the K9 Service, which routes a request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetAssignmentSummary). The API retrieves the current handler assignment and determines whether a previous handler exists. This information is used to display the assigned handler and determine which handler action buttons (Assign, Update, Deactivate, or Reactivate) are available. **K9User Entity:** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate

			User Entity: • UserId (PK) • UserName • UserSurname • ActiveStatus
			Step 7: The system displays a View K9 Details modal on the Angular frontend containing the selected K9's details. The modal is triggered when a K9 card is selected. The View K9 Details modal contains the following components: • Modal heading: ○ K9 name • K9 image • K9 details section: ○ Breed ○ Gender ○ Age ○ Date of Birth ○ Father ○ Mother ○ Status ○ Assigned K9 Roles ○ Assigned Handler (or "No Handler Assigned" if none exists) • Handler Actions:

			○ Assign or Update Handler button (depending on whether a current handler exists) ○ Deactivate Handler button (displayed when a current handler exists) ○ Reactivate Handler button (displayed when no current handler exists but a previous handler exists) • Action buttons section: ○ View Ancestry button ○ Edit button ○ Deactivate button ○ Close button
	Step 8: The handler can view the selected K9's details.		
Alternate courses:	**ALT-Step 3:** The handler does not search or filter for a dog and just selects the dog directly from the ones displayed. Go to Step 6.		
	ALT-Step 4: The system does not find any dogs matching the specified criteria. Go back to step 2.		
Conclusion:	The system displays a modal showing the dog's details.		
Post-condition:	The dog's details are displayed.		
Business rules:	• Only super admins or users given appropriate permissions can view a dog.		

Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 92: View Dog - Technical Use Case Narrative

2.8.3 View Dogs

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">View Dogs</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.3</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Handler</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a handler views all K9s registered in the system. The system retrieves the K9 records from the database and displays them as K9 cards on the K9 page. The handler can then search for K9s by name or breed, filter the displayed K9s by gender or status, and select a K9 to view its detailed information.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The handler must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The handler wants to view the specific dog’s details on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The handler wants to view all K9s on the system.</td><td></td><td></td></tr>
<tr><td>Step 2: The handler selects the “Animals” tab to navigate to the animals’ page. [ALT]</td><td></td><td>Step 3: The system displays the Animals page containing:
• A secondary navigation bar with the following animal category tabs:</td></tr>
</table>

			○ Rhino ○ Elephant ○ K9 • A content area below the navigation bar used to display the selected animal records. • If no animal category has been selected, the content area displays the following message: ○ "Select an animal category" ○ "Choose Rhino, Elephant, or K9 to view records."
	Step 4: The handler then selects “K9” on the secondary navbar to navigate to the K9s page.		**Step 5:** The system retrieves all K9 records using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetAllK9s). The API retrieves data from the following entity: **Animal Entity:** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalGender • AnimalDateOfBirth • AnimalPhoto

			• ActiveStatus **K9 Entity:** • K9Id (PK) • AnimalId (FK) • K9DadId (FK) • K9MomId (FK)
			Step 6: The system displays the retrieved K9s as K9 cards on the Angular frontend. Each K9 card displays the following information: • Image • Name • Age • Gender • Breed The page also provides the following filtering options: • Search by K9 name or breed. • Filter by gender. • Filter by status (Active, Inactive, or All). • Results counter
	Step 7: The Super Admin can view the K9s.		
Alternate courses:	**ALT-Step 2:** User is on mobile and selects the menu icon. Go to step 3.		
Conclusion:	The system displays all the dogs on the system.		

Post-condition:	All the dogs on the system are displayed.
Business rules:	• Only super admins or users given appropriate permissions can view dogs.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 93: View Dogs - Technical Use Case Narrative

2.8.4 Update Dog

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Update Dog</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.4</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a super admin updates the details of an existing K9 within the system. The system allows the super admin to modify the K9's information, update its assigned roles, change its parent information, and optionally replace the K9's image. The system validates the updated information, requests confirmation before saving the changes, updates the K9 record and its associated role assignments in the database, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to update a dog’s details on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to update a K9's details on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display the View K9 Details modal containing the selected K9's information. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint</td></tr>
</table>

			in Visual Studio (K9Controller – GetK9ById).
	Step 3: The super admin selects the "Edit" button. [ALT]		**Step 4:** The system loads and displays the Edit K9 modal with the selected K9's current information. The Edit K9 modal contains the following components: • Edit K9 modal title. • Name section: ○ Name label. ○ Name input field. • Breed section: ○ Breed label. ○ Breed input field. • Gender section: ○ Gender label. ○ Dropdown list. ▪ Male. ▪ Female. • K9 Roles section: ○ K9 Roles label. ○ Multi-select dropdown displaying all available K9 Roles. ○ Selected role counter. • Date of Birth section: ○ Date of Birth label. ○ Date selector. • Father section:

			○ Father label. ○ Dropdown list. ▪ Default option: "No Father Assigned". ▪ Displays all available male K9s except the K9 currently being edited. • Mother section: ○ Mother label. ○ Dropdown list. ▪ Default option: "No Mother Assigned". ▪ Displays all available female K9s except the K9 currently being edited. • Current Image section: ○ Displays the K9's current image. • Replace Image section: ○ Choose Image button. ○ Displays the selected replacement image file name. • Action buttons: ○ Save Changes button.

			○ Cancel button. The Father and Mother dropdown lists are populated using the K9 data that was previously retrieved when UC 8.2 (View K9 Details) was invoked. The K9 Roles dropdown is populated using data retrieved from the K9 Role Entity. **K9 Role Entity** • K9RoleId (PK) • RoleName
	Step 5: The super admin updates the required K9 details and selects the "Save Changes" button. [ALT]		**Step 6:** The system validates the updated information using Angular frontend validation before allowing the K9 to be updated. The system validates the following: • The K9 Name field is not null or empty. • The K9 Name contains alphabetic characters only. • The Breed field is not null or empty. • The Breed contains alphabetic characters only. • The Date of Birth is not in the future. • If a father is selected, the father must be at least two years older than the K9. • If a mother is selected, the mother must be at least two years older than the K9. • If a replacement image is selected, only JPG, JPEG, PNG

			or WEBP image formats are accepted. [ALT]
			Step 7: The system displays a custom Angular confirmation modal. The confirmation modal contains: • Confirmation modal title ("Confirm Update"). • Confirmation message label: ○ Displays the message: "Are you sure you want to update this K9?" • Action buttons: ○ Confirm button. ○ Cancel button.
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system captures the updated K9 information and constructs a FormData object containing the modified K9 details, selected K9 Roles, optional replacement image, and parent information. The Angular frontend uses the K9 Service and Angular HTTP Client to route the request to the corresponding HTTP PUT API endpoint in Visual Studio (K9Controller – UpdateK9). The API passes the request to the K9 Repository, where additional business validation is performed before the updated information is saved using Entity Framework Core in Microsoft SQL Server.

			The backend validates that: • A K9 cannot be assigned as its own parent. • The selected K9 is active and may be modified. • The selected father and mother cannot be the same K9. • The selected father exists and is male. • The selected mother exists and is female. • The selected parents are each at least two years older than the K9. If a replacement image is supplied, the existing image is removed and the new image is uploaded and stored. The system updates the associated Animal record and K9 record, and the K9's role assignments are updated in the K9K9Role junction table before the changes are saved to the database. **Animal Entity** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalGender • AnimalDateOfBirth • AnimalPhoto • ActiveStatus

			K9 Entity • K9Id (PK) • AnimalId (FK) • K9DadId (FK) • K9MomId (FK) **K9K9Role Entity (Junction Table)** • K9Id (PK, FK) • K9RoleId (PK, FK) **K9 Role Entity** • K9RoleId (PK) • RoleName [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the K9. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that the K9 has been updated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a toaster notification confirming that the K9 has been successfully updated.

			The confirmation message displays the text: "K9 updated successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the "Edit" button. Terminate use case.		
	ALT-Step 5: The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 6: The system finds an error after validating and an error message is displayed within the Edit K9 modal. Go back to step 4.		
	ALT-Step 8: The super admin does not confirm and selects the "Cancel" button. Go back to step 4.		
	ALT-Step 9: The system fails to save the changes and displays a toaster message: "Failed to update K9". Terminate use case.		
Conclusion:	The new dog's details have been saved onto the system, and a confirmation message is shown.		
Post-condition:	The system saves the new K9 details in the Animal and K9 table, stores any associated new K9 role assignments, uploads and stores the selected image (if changed), and makes the newly updated K9's details available for viewing and management within the system.		
Business rules:	• Only super admins or users given appropriate permissions can update a dog.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 94: Update Dog - Technical Use Case Narrative

2.8.5 Deactivate Dog

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Deactivate Dog</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.5</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin deactivates an existing K9. The system displays the selected K9's details, requests confirmation before proceeding, updates the K9's status to inactive, records the action in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to deactivate a dog on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to deactivate a dog on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display a pop-up showing the selected K9's details. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9ById).</td></tr>
<tr><td>Step 3: The super admin selects the</td><td></td><td>Step 4: The system displays a custom Angular confirmation modal, which</td></tr>
</table>

	"Deactivate" button. [ALT]		is triggered when the Super Admin clicks the Deactivate button. • Confirmation modal title ("Confirm Deactivation") • Confirmation message label ○ Displays the message: "Are you sure you want to deactivate this K9?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system deactivates the selected K9 by routing the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9Controller – DeactivateK9). The API validates that the K9 exists and performs a soft delete by updating the ActiveStatus field of the associated Animal record to false using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process: **Animal Entity** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalGender

			• AnimalDateOfBirth • AnimalPhoto • ActiveStatus **K9 Entity** • K9Id (PK) • AnimalId (FK) • K9DadId (FK) • K9MomId (FK) The deactivated K9 is no longer displayed in the active K9 listings. [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the K9. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a K9 has been deactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a toaster notification confirming that the K9 has been successfully deactivated.

			The confirmation message displays the text: "K9 deactivated successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the "Deactivate" button. Terminate use case.		
	ALT-Step 5: The super admin does not confirm and selects the "Cancel" button. Go back to step 2.		
	ALT-Step 6: The system fails to deactivate the K9 and displays a toaster message: "Failed to deactivate K9". Terminate use case.		
Conclusion:	The K9 has been deactivated on the system, and a confirmation message is shown.		
Post-condition:	The system sets the K9's status to inactive in the Animal table and the K9 is then no longer displayed in the active K9 listings.		
Business rules:	• Only super admins or users given appropriate permissions can deactivate a dog.		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 95: Deactivate Dog - Technical Use Case Narrative

2.8.6 Assign User to Dog

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Assign User to Dog</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.6</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• Handler (ERA)</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin assigns a handler to a selected dog. The system displays an assignment form that allows the super admin to search for and select an active handler. Once the assignment is confirmed, the system validates the request, creates a new handler assignment linked to the selected dog, notifies the assigned handler, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to assign a handler to a dog on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to assign a handler to a dog on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display a pop-up showing the selected K9's details. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9ById).</td></tr>
</table>

	Step 3: The super admin selects the "Assign" button. [ALT]		**Step 4:** The system displays a custom Angular Assign Handler modal, allowing the Super Admin to assign a handler to the selected K9. The assign handler modal contains the following components: • Modal heading ("Assign Handler") • Handler selection section: ○ Search textbox for locating handlers ○ Auto-complete list displaying matching active handlers • Action buttons: ○ Assign button ○ Cancel button The User search list is populated using data retrieved from the User Entity via the User Service, which calls the backend API. **User Entity** • UserId (PK) • UserName • UserSurname • ActiveStatus
	Step 5: The super admin searches for and selects a handler from the list before selecting the		**Step 6:** The system displays a custom Angular confirmation modal requesting confirmation before assigning the selected handler to the K9.

	"Assign" button. [ALT]		The confirmation modal contains the following components: • Confirmation modal title ("Confirm Handler Assignment") • Confirmation message label ○ Displays the message: "Are you sure you want to assign this handler to the selected K9?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 7: The super admin selects the "Confirm" button. [ALT]		**Step 8:** The system creates a new handler assignment by routing the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP POST API endpoint in Visual Studio (K9Controller – AssignUserToDog). The API validates that the selected K9 exists and is active, verifies that the selected user exists and is active, and confirms that the K9 does not already have an active handler. Entity Framework Core then creates a new assignment record in MS SQL Server. The following entity is involved in the process: **Animal Entity** • AnimalId (PK) • ActiveStatus

			K9 Entity • K9Id (PK) • AnimalId (FK) **K9User Entity** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate **User Entity** • UserId (PK) • UserName • UserSurname • ActiveStatus The assignment record stores the current date and time as the StartDate, while the EndDate remains null until the assignment is deactivated or updated. [ALT]
			Step 9: The system notifies the assigned handler of the new assignment. This notification is handled by the system's email notification service, which sends an email informing the handler that they have been assigned to the selected K9.
			Step 10: The system records the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the handler assignment.

			The system stores information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a handler has been assigned to a K9. **AuditLog Entity** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a toaster notification confirming that the handler has been successfully assigned. The confirmation message displays the text: "Handler assigned successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the "Assign" button. Terminate use case.		
	ALT-Step 5: The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		
	ALT-Step 7: The super admin does not confirm and selects the "Cancel" button. Go back to step 4.		
	ALT-Step 8: The system fails to create the user-dog assignment and displays a toaster message: "Failed to assign handler". Terminate use case.		
Conclusion:	The dog has been assigned to a handler on the system, and a confirmation message is shown.		
Post-condition:	A new handler assignment is created in the K9User table, linking the selected handler to the selected dog, and the assignment becomes the dog's current active handler.		

Business rules:	• Only super admin or users given appropriate permissions can assign a handler to a dog. • A dog can only have one handler.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 96: Assign User to Dog - Technical Use Case Narrative

2.8.7 View User-Dog Assignment

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">View User-Dog Assignment</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.7</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Handler</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how a handler views all K9 handler assignments recorded in the system. The system retrieves all K9 and handler assignment records from the database and displays them in a table, allowing the handler to search by K9 or handler name and filter assignments by their status (Active, Inactive, or All).</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The handler must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The handler wants to view all the dogs and their current handlers on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The handler wants to view all K9 handler assignments on the system.</td><td></td><td>Step 2: The system invokes UC 8.3 (View Dogs) to display the K9 page showing all registered K9s. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetAllK9s).</td></tr>
</table>

	Step 3: The super admin selects the "View Assignments " button.		**Step 4:** The Angular frontend navigates to the K9 Assignments page. The system then retrieves the assignments through the K9 Service in the Angular frontend, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9UserAssignments). The API retrieves all K9 handler assignments from the database using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process: **Animal Entity** • AnimalId (PK) • AnimalName **K9 Entity** • K9Id (PK) • AnimalId (FK) **K9User Entity** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate **User Entity** • UserId (PK) • UserName • UserSurname

			• ActiveStatus
			Step 5: The system displays the K9 Assignments page containing a table of all handler assignments. The table displays the following information: • K9 Name • Handler Name • Start Date • End Date • Assignment Status (Active/Inactive) The page also provides the following filtering options: • Search by K9 name. • Search by handler name. • Filter by assignment status (Active, Inactive, or All). [ALT]
	Step 6: The handler can view the K9 handler assignments.		
Alternate courses:	**ALT-Step 5**: There are currently no assignments, and the system displays a message to the handler. Terminate use case		
Conclusion:	The system displays all the assignments on the system.		
Post-condition:	The current K9 handler assignments are displayed on the K9 Assignments page, and the handler can view, search, and filter the assignment records.		
Business rules:	• Only super admins or users given appropriate permissions can view assignments.		

Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 97: View User-Dog Assignment - Technical Use Case Narrative

2.8.8 Update User-Dog Assignment

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Update User-Dog Assignment</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.8</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• Handler (ERA)</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin updates the handler assigned to a selected dog. The system displays an update handler form that allows the super admin to search for and select a different active handler. Once the update is confirmed, the system validates the request, ends the current handler assignment, creates a new handler assignment linked to the selected dog, notifies both the previously assigned and newly assigned handlers via email, records the transaction in the AuditLog, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to change a dog’s handler on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to update a dog's handler assignment on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display a pop-up showing the selected K9's details. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API</td></tr>
</table>

			endpoint in Visual Studio (K9Controller – GetK9ById).
	Step 3: The super admin selects the "Update" button. [ALT]		**Step 4:** The system displays a custom Angular Update Handler modal, allowing the Super Admin to assign a different handler to the selected K9. The update handler modal contains the following components: • Modal heading ("Update Handler") • Handler selection section: ○ Search textbox for locating handlers ○ Auto-complete list displaying matching active handlers ○ The currently assigned handler is pre-selected • Action buttons: ○ Update button ○ Cancel button The User search list is populated using data retrieved from the User Entity via the User Service, which calls the backend API. **User Entity** • UserId (PK) • UserName • UserSurname • ActiveStatus
	Step 5: The super admin searches for		**Step 6:** The system displays a custom Angular confirmation modal

	and selects a new handler, then selects the “Update” button. [ALT]		requesting confirmation before updating the handler assignment. The confirmation modal contains: • Modal heading ("Confirm Handler Update") • Confirmation message ○ "Are you sure you want to update this handler assignment?" • Action buttons: ○ Confirm ○ Cancel
	Step 7: The super admin selects the “Confirm” button. [ALT]		**Step 8:** The system routes the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9Controller – UpdateUserDogAssignment). The API validates that the selected K9 exists and is active, verifies that an active assignment exists for the selected K9, confirms that the newly selected handler is different from the current handler, and validates that the selected user exists and is active. Entity Framework Core then closes the current assignment by recording the current date and time as the EndDate and creates a new assignment record with the newly selected handler and the current date and time as the StartDate.

			The following entities are involved in the process: **Animal Entity** • AnimalId (PK) • ActiveStatus **K9 Entity** • K9Id (PK) • AnimalId (FK) **K9User Entity** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate **User Entity** • UserId (PK) • UserName • UserSurname • ActiveStatus [ALT]
			Step 9: The system notifies the previously assigned handler that they are no longer assigned to the selected K9. This notification is handled by the system's email notification service, which sends an email informing the handler that the assignment has been removed.
			Step 10: The system notifies the newly assigned handler of the updated assignment. This notification is

			handled by the system's email notification service, which sends an email informing the handler that they have been assigned to the selected K9.
			Step 11: The system records the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the handler assignment. The system stores information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a handler has been assigned to a K9. **AuditLog Entity** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 12: The system displays a toaster notification confirming that the handler assignment has been successfully updated. The confirmation message displays the text: "Handler updated successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the "Update" button. Terminate use case.		
	ALT-Step 5: The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.		

	ALT-Step 7: The super admin does not confirm and selects the "Cancel" button. Go back to step 4.
	ALT-Step 8: The system fails to create the update the user-dog assignment and displays a toaster message: "Failed to update handler". Terminate use case.
Conclusion:	The dog has been assigned to a handler on the system, and a confirmation message is shown.
Post-condition:	The current handler assignment is updated successfully. The previous assignment is marked as inactive, a new active assignment is created for the selected handler, notification emails are sent to both affected handlers, and the updated assignment is available for viewing on the system.
Business rules:	• Only super admins or users given appropriate permissions can change a dog's handler. • A dog can only have one handler.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 98: Update User-Dog Assignment - Technical Use Case Narrative

2.8.9 Reactivate User-Dog Assignment

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Reactivate User-Dog Assignment</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.9</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• Handler (ERA)</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin restores a previous handler assignment for a selected dog. The system allows the super admin to reactivate the most recent previous handler assignment after confirmation. The system validates the request, creates a new active assignment for the previous handler, notifies the restored handler via email, records the transaction in the AuditLog, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to reactivate a user-dog assignment on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to reactivate a previous handler assignment for a selected</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display a pop-up showing the selected K9's details. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9ById).</td></tr>
</table>

	dog on the system.		
	Step 3: The super admin selects the "Reactivate" button. [ALT]		**Step 4:** The system displays a custom Angular confirmation modal requesting confirmation before restoring the previous handler assignment. The confirmation modal contains: • Modal heading ("Restore Previous Handler") • Confirmation message ○ "Are you sure you want to restore the previous handler assignment for this K9?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system routes the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9Controller – ReactivateUserDogAssignment). The API validates that the selected K9 exists and is active. It then retrieves the most recent previous handler assignment from the database. Entity Framework Core creates a new assignment record for the previous handler, recording the current date and time as the StartDate and leaving the EndDate as null.

			The following entities are involved in the process: **Animal Entity** • AnimalId (PK) • ActiveStatus **K9 Entity** • K9Id (PK) • AnimalId (FK) **K9User Entity** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate **User Entity** • UserId (PK) • UserName • UserSurname • ActiveStatus [ALT]
			Step 7: The system notifies the restored handler that they have been assigned to the selected K9. This notification is handled by the system's email notification service, which sends an email informing the handler that their assignment has been restored.
			Step 8: The system records the transaction in the AuditLog table to maintain a history of actions performed within the system. This is

			handled in the backend after the successful reactivation of the previous handler assignment. The system stores information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a handler has been assigned to a K9. **AuditLog Entity** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a toaster notification confirming that previous handler assignment has been successfully reactivated. The confirmation message displays the text: "Previous handler restored successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the “Reactivate” button. Terminate use case.		
	ALT-Step 5: The super admin does not confirm and selects the “Cancel” button. Go back to step 2.		
	ALT-Step 6: The system fails to reactivate the user-dog assignment and displays a toaster message: “Failed to restore handler”. Terminate use case.		
Conclusion:	The dog’s previous assignment has been reactivated, and a confirmation message is shown.		
Post-condition:	The previous handler assignment is restored successfully. A new active assignment is created for the previous handler, the restored handler is		

	notified via email, and the updated assignment is available for viewing on the system.
Business rules:	• Only super admins or users given appropriate permissions can reactivate a user-dog assignment
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 99: Reactivate User-Dog Assignment - Technical Use Case Narrative

2.8.10 Deactivate User-Dog Assignment

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Deactivate User-Dog Assignment</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.10</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• Handler (ERA)</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin removes the current handler assignment from a selected dog. The system requests confirmation before removing the assignment. Once confirmed, the system validates the request, deactivates the current user-dog assignment by recording an end date, notifies the removed handler, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to deactivate a user-dog assignment on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to deactivate a user-dog assignment on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display a pop-up showing the selected K9's details. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9ById).</td></tr>
</table>

	Step 3: The super admin selects the "Deactivate" button. [ALT]		**Step 4:** The system displays a custom Angular confirmation modal requesting confirmation before restoring the previous handler assignment. The confirmation modal contains: • Modal heading ("Confirm Handler Removal") • Confirmation message ○ "Are you sure you want to remove the current handler from this K9?" • Action buttons: ○ Confirm button ○ Cancel button
	Step 5: The super admin selects the "Confirm" button. [ALT]		**Step 6:** The system routes the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9Controller – DeactivateUserDogAssignment). The API validates that the selected K9 exists and is active. It retrieves the current active handler assignment and records the current date and time as the EndDate, thereby marking the assignment as inactive. Entity Framework Core then saves the updated assignment in Microsoft SQL Server. The following entity is involved in the process:

			Animal Entity • AnimalId (PK) • ActiveStatus **K9 Entity** • K9Id (PK) • AnimalId (FK) **K9User Entity** • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate [ALT]
			Step 7: The system notifies the previously assigned handler that they are no longer assigned to the selected K9. This notification is handled by the notification component of the system and informs the handler that their assignment has been removed.
			Step 8: The system records the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the previous handler assignment. The system stores information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a handler has been removed from a K9. **AuditLog Entity**

			• AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 9: The system displays a toaster notification confirming that the current handler assignment has been successfully deactivated. The confirmation message displays the text: "Handler removed successfully."
Alternate courses:	**ALT-Step 3:** The selected K9 is inactive, the Angular frontend does not display the "deactivate" button. Terminate use case.		
	ALT-Step 5: The super admin does not confirm and selects the "Cancel" button. Go back to step 2.		
	ALT-Step 6: The system fails to deactivate the user-dog assignment and displays a toaster message: "Failed to remove handler". Terminate use case.		
Conclusion:	The dog's current assignment has been deactivated, and a confirmation message is shown.		
Post-condition:	The current user-dog assignment has been deactivated. The selected dog no longer has an active handler assignment, the previous handler has been notified of the removal, and the updated assignment is available for viewing on the system.		
Business rules:	• Only super admins or users given appropriate permissions can deactivate a user-dog assignment		
Implementation constraints and specifications:	• None		
Assumptions:	• None		
Open issues:	• None		

Table 100: Deactivate User-Dog Assignment - Technical Use Case Narrative

2.8.11 Add K9 Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Add K9 Role</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.11</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin creates a new K9 role within the system. The system displays a form that allows the super admin to enter the role name and description. Once the information is validated and confirmed, the system creates the new K9 role, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to add a new K9 role to the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The super admin wants to add a new K9 role to the system.</td><td></td><td>Step 2: The system invokes UC 8.13 (View K9 Roles) to display the K9 Roles page showing all registered K9 roles. This is done using the Angular frontend and the K9 Role Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9RoleController – GetAllK9Roles).</td></tr>
</table>

	Step 3: The super admin selects the "Create Role" button.		**Step 4:** The system loads and displays the Create K9 Role modal with the following components: • Create K9 Role modal title. • Role Name section: ○ Role Name label. ○ Role Name input field. ○ Placeholder text: *"e.g. Tracker"*. • Role Description section: ○ Role Description label. ○ Role Description input field. ○ Placeholder text: "What does this role do?". • Action buttons: ○ Create button. ○ Cancel button. The Create button remains disabled until both the Role Name and Role Description fields contain values.
	Step 5: The super admin enters the K9 role details and selects the "Create" button. [ALT]		**Step 6:** The system validates the entered details using Angular frontend validation before allowing the K9 role to be created. The system validates the following: • The Role Name field is not null or empty. • The Role Description field is not null or empty. [ALT]

			Step 7: The system displays a custom Angular confirmation modal after successful validation. The confirmation modal contains: • Confirmation modal title ("Confirm Create Role"). • Confirmation message label: ○ Displays the message: ▪ "Are you sure you want to create this role?" • Displays the name of the new role for confirmation. • Action buttons: ○ Confirm button. ○ Cancel button.
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system creates a new K9 Role by routing the request through the Angular frontend and K9 Role Service, which sends the request via the Angular HTTP Client to the corresponding HTTP POST API endpoint in Visual Studio (K9RoleController – CreateK9Role). The API endpoint passes the request to the K9 Role Repository, where a new K9Role entity is created. The repository automatically assigns the ActiveStatus value as true before the data is saved using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process:

			K9 Role Entity • K9RoleId (PK) • RoleName • RoleDescription • ActiveStatus After the entity has been created, Entity Framework Core generates the primary key and stores the new K9 Role record in the K9Role table. [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the new K9 Role. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a new K9 Role has been created. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a toaster notification confirming that the K9 Role has been successfully created. The confirmation message displays the text: "Role created successfully."

Alternate courses:	**ALT-Step 5:** The super admin changes their mind and closes the pop-up by selecting the "Cancel" button. Terminate use case.
	ALT-Step 6: The system finds an error after validating and an error message is displayed within the Create K9 Role modal. Go back to step 4.
	ALT-Step 8: The super admin does not confirm and selects the "Cancel" button. Go back to step 4.
	ALT-Step 9: The system fails to save the new K9 role and displays a toaster message: "Failed to create role". Terminate use case.
Conclusion:	The new K9 role's details have been saved onto the system, and a confirmation message is shown.
Post-condition:	The system saves the new K9 Role in the K9Role table and makes the newly created role available for assignment to K9s and for management within the system.
Business rules:	• Only super admin or users given appropriate permissions can add a new K9 role.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 101: Add K9 Role - Technical Use Case Narrative

2.8.12 Update K9 Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Update K9 Role</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.12</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin updates an existing K9 role within the system. The system displays the selected K9 role's details in an editable form, allowing the super admin to modify the role name and description. Once the updated information is validated and confirmed, the system saves the changes, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to update a K9 role’s details on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to update a K9 role’s detail on the system.</td><td></td><td>Step 2: The system invokes UC 8.13 (View K9 Roles) to display the K9 Roles page showing all registered K9 roles. This is done using the Angular frontend and the K9 Role Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9RoleController – GetAllK9Roles).</td></tr>
</table>

	Step 3: The super admin selects the "Edit" button for a specific K9 Role.		**Step 4:** The system loads and displays the Edit K9 Role modal with the selected role's existing details pre-populated. The Edit K9 Role modal contains the following components: • Edit K9 Role modal title. • Role Name section: ○ Role Name label. ○ Role Name input field containing the current role name. • Role Description section: ○ Role Description label. ○ Role Description input field containing the current role description. • Action buttons: ○ Save button. ○ Cancel button. The Save button remains disabled until both the Role Name and Role Description fields contain values.
	Step 5: The super admin updates the K9 Role details and selects the "Save" button. [ALT]		**Step 6:** The system validates the entered details using Angular frontend validation before allowing the K9 Role to be updated. The system validates the following: • The Role Name field is not null or empty. • The Role Description field is not null or empty.

			[ALT]
			Step 7: The system displays a custom Angular confirmation modal after successful validation. The confirmation modal contains: • Confirmation modal title ("Confirm Update Role"). • Confirmation message label: ○ Displays the message: ▪ "Are you sure you want to update this role?" • Displays the updated role name for confirmation. • Action buttons: ○ Confirm button. ○ Cancel button.
	Step 8: The super admin selects the "Confirm" button. [ALT]		**Step 9:** The system updates the selected K9 Role by routing the request through the Angular frontend and K9 Role Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9RoleController – UpdateK9Role). The API endpoint passes the request to the K9 Role Repository, where the selected K9 Role is retrieved from the database using its primary key. The repository updates the Role Name and Role Description before saving the changes using Entity Framework Core in Microsoft SQL Server.

			The following entity is involved in the process: **K9 Role Entity** • K9RoleId (PK) • RoleName • RoleDescription • ActiveStatus [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the K9 Role. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a K9 Role has been updated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a toaster notification confirming that the K9 Role has been successfully updated. The confirmation message displays the text: "Role updated successfully."
Alternate courses:	**ALT-Step 5:**		

	The super admin changes their mind and closes the pop-up by selecting the “Cancel” button. Terminate use case.
	ALT-Step 6: The system finds an error after validating and an error message is displayed within the Edit K9 Role modal. Go back to step 4.
	ALT-Step 8: The super admin does not confirm and selects the “Cancel” button. Go back to step 4.
	ALT-Step 9: The system fails to save the changes and displays a toaster message: “Failed to update role”. Terminate use case.
Conclusion:	The updated K9 role’s details have been saved onto the system, and a confirmation message is shown.
Post-condition:	The system saves the updated K9 Role details in the K9Role table, and the modified role is immediately available for viewing, assignment to K9s, and management within the system.
Business rules:	• Only super admin or users given appropriate permissions can update a K9 role.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 102: Update K9 Role - Technical Use Case Narrative

2.8.13 View K9 Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">View K9 Roles</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.13</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Handler</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the handler views the K9 Roles available within the system. The system retrieves all K9 Roles from the database and displays them in a searchable and filterable table showing the role name, description, status, and available actions.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The handler must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The handler wants to view the K9 roles on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The handler wants to view the K9 roles on the system.</td><td></td><td>Step 2: The system invokes UC 8.3 (View Dogs) to display the K9 page showing all registered K9s. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetAllK9s).</td></tr>
</table>

	Step 3: The handler selects the "Manage K9 Roles" button.		**Step 4:** The system retrieves the K9 Roles by routing the request through the Angular frontend and K9 Role Service, which sends the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9RoleController – GetAllK9Roles). The API endpoint passes the request to the K9 Role Repository, where all K9 Roles are retrieved from the database using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process: **K9 Role Entity** • K9RoleId (PK) • RoleName • RoleDescription • ActiveStatus The repository returns the collection of K9 Roles to the controller, which returns the data to the Angular frontend.
			Step 5: The system displays the retrieved K9 Roles in a table on the Angular frontend. Each K9 Role displays the following information:

			• Role Name • Role Description • Status • Available Actions The page also provides the following filtering options: • Search by role name. • Filter by status (Active, Inactive, or All). • Results counter. Each K9 Role also provides the following action buttons: • Edit. • Deactivate (for active roles). • Reactivate (for inactive roles). If no K9 Roles are available, the system displays the message: "No roles found."
	Step 6: The handler can view the K9 roles		
Alternate courses:	None		
Conclusion:	The system displays a pop-up with a table showing all the current roles.		
Post-condition:	The system retrieves and displays the K9 Roles stored in the K9Role table, allowing authorized users to view, search, and filter the available roles within the system.		
Business rules:	• Only super admin or users given appropriate permissions can view K9 roles.		
Implementation constraints and specifications:	• None		

Assumptions:	• None
Open issues:	• None

Table 103: View K9 Roles - Technical Use Case Narrative

2.8.14 Reactivate K9 Role

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 30/04/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Reactivate K9 Role</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.14</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the super admin reactivates an inactive K9 Role within the system. The system requests confirmation before restoring the role's active status. Once confirmed, the system updates the K9 Role, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin wants to reactivate a K9 role on the system.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to reactivate a K9 Role on the system</td><td></td><td>Step 2: The system invokes UC 8.13 (View K9 Roles) to display the K9 Roles page showing all registered K9 Roles. This is done using the Angular frontend and the K9 Role Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9RoleController – GetAllK9Roles).</td></tr>
</table>

	Step 3: The super admin selects the "Reactivate" button for a K9 Role with an inactive status.		**Step 4:** The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the Reactivate button. • Confirmation modal title ("Confirm Reactivate Role") • Confirmation message label ○ Displays the message: ▪ "Are you sure you want to reactivate this role?" • Displays the selected K9 Role name. • Action buttons: ○ Reactivate button. ○ Cancel button.
	Step 5: The super admin selects the "Reactivate" button. [ALT]		**Step 6:** The system reactivates the selected K9 Role by routing the request through the Angular frontend and K9 Role Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9RoleController – ReactivateK9Role). The API validates that the K9 Role exists and updates the ActiveStatus field to true using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process: **K9 Role Entity** • K9RoleId (PK)

			• RoleName • RoleDescription • ActiveStatus The reactivated K9 Role is displayed in the active K9 Role listings and becomes available for assignment to K9s. [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful reactivation of the K9 Role. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a K9 Role has been reactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a toaster notification confirming that the K9 Role has been successfully reactivated. The confirmation message displays the text: "Role reactivated successfully."
Alternate courses:	**ALT-Step 5:**		

	The super admin does not confirm and selects the "Cancel" button. Go back to step 2.
	ALT-Step 6: The system fails to reactivate the K9 role and displays a toaster message: "Failed to reactivate role". Terminate use case.
Conclusion:	The inactive K9 role has been reactivated, and a confirmation message is shown
Post-condition:	The system updates the K9 Role's status to active in the K9Role table, making the role available for viewing, assignment to K9s, and management within the system.
Business rules:	• Only super admin or users given appropriate permissions can reactivate a K9 role.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 104: Reactivate K9 Role - Technical Use Case Narrative

2.8.15 Deactivate K9 Role

<table>
<tr><th colspan="5">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td colspan="2">Date: 30/04/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">Deactivate K9 Role</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.15</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="4">Super Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="4">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="4">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="4">• None</td></tr>
<tr><td>Description:</td><td colspan="4">This use case describes how the super admin deactivates an active K9 Role within the system. The system requests confirmation before deactivating the role's active status. Once confirmed, the system updates the K9 Role, records the transaction in the audit log, and displays a confirmation message.</td></tr>
<tr><td>Pre-condition:</td><td colspan="4">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="4">The super admin wants to deactivate a K9 role on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="3">System Response</td></tr>
<tr><td>Manual Action</td><td colspan="2">Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to deactivate a K9 Role on the system.</td><td></td><td colspan="2">Step 2: The system invokes UC 8.13 (View K9 Roles) to display the K9 Roles page showing all registered K9 Roles. This is done using the Angular frontend and the K9 Role Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9RoleController – GetAllK9Roles).</td></tr>
</table>

	Step 3: The super admin selects the "Deactivate" button for a K9 Role with an active status.		**Step 4:** The system displays a custom Angular confirmation modal, which is triggered when the Super Admin clicks the Deactivate button. • Confirmation modal title ("Confirm Deactivate Role") • Confirmation message label ○ Displays the message: ▪ "Are you sure you want to deactivate this role?" • Displays the selected K9 Role name. • Action buttons: ○ Deactivate button. ○ Cancel button.
	Step 5: The super admin selects the "Deactivate" button. [ALT]		**Step 6:** The system deactivates the selected K9 Role by routing the request through the Angular frontend and K9 Role Service, which sends the request via the Angular HTTP Client to the corresponding HTTP PUT API endpoint in Visual Studio (K9RoleController – DeactivateK9Role). The API validates that the K9 Role exists and updates the ActiveStatus field to false using Entity Framework Core in Microsoft SQL Server. The following entity is involved in the process: **K9 Role Entity** • K9RoleId (PK)

			• RoleName • RoleDescription • ActiveStatus The deactivated K9 Role is no longer displayed in the active K9 Role listings but remains available when viewing all or inactive K9 Roles. [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the K9 Role. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a K9 Role has been deactivated. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a toaster notification confirming that the K9 Role has been successfully deactivated. The confirmation message displays the text: "Role deactivated successfully."
Alternate courses:	**ALT-Step 5:**		

	The super admin does not confirm and selects the "Cancel" button. Go back to step 2.
	ALT-Step 6: The system fails to deactivate the K9 role and displays a toaster message: "Failed to deactivate role". Terminate use case.
Conclusion:	The active K9 role has been deactivated, and a confirmation message is shown.
Post-condition:	The system updates the K9 Role's status to inactive in the K9Role table, making the role unavailable in the active K9 Role listings while remaining available for viewing and management when inactive roles are displayed.
Business rules:	• Only super admin or users given appropriate permissions can deactivate a K9 role.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 105: Deactivate K9 Role - Technical Use Case Narrative

2.8.16 View K9 Ancestry

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Nicholas Bailey</td><td>Date: 13/07/2026
Version: 1</td></tr>
<tr><td>Use case name:</td><td colspan="2">View K9 Ancestry</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">8.16</td></tr>
<tr><td>Priority:</td><td colspan="2">Low</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Handler</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how the handler views the ancestry of a selected K9 within the system. The system retrieves the selected K9's lineage information from the database and displays it as a graphical ancestry tree showing the current K9, its parents, grandparents, and great-grandparents.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The handler must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The handler wants to view the ancestry tree of the dogs on the system.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The handler wants to view the ancestry tree of the dogs on the system.</td><td></td><td>Step 2: The system invokes UC 8.2 (View Dog) to display the View K9 Details modal containing the selected K9's information. This is done using the Angular frontend and the K9 Service, which routes the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9ById).</td></tr>
</table>

	Step 3: The super admin selects the "View Ancestry" button.		**Step 4:** The system retrieves the selected K9's ancestry by routing the request through the Angular frontend and K9 Service, which sends the request via the Angular HTTP Client to the corresponding HTTP GET API endpoint in Visual Studio (K9Controller – GetK9Ancestry). The API endpoint passes the request to the K9 Repository, where the selected K9 and its lineage are retrieved using Entity Framework Core in Microsoft SQL Server. The repository recursively traverses the parent relationships stored in the K9 table to construct the ancestry tree for the requested number of generations. The following entity is involved in the process: **Animal Entity** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalPhoto **K9 Entity** • K9Id (PK) • AnimalId (FK) • K9DadId (FK) • K9MomId (FK) [ALT]
			Step 5: The system displays the retrieved ancestry information as a

			horizontal graphical pedigree tree on the Angular frontend. The ancestry tree displays the following information for each available K9: • Image • Name • Breed The ancestry tree displays the following generations: • Current K9 • Parents • Grandparents • Great-grandparents Where no parent information exists, the system displays "Unknown" in the corresponding position. The Angular frontend also draws connector lines between related K9s and highlights the direct ancestral lineage when the handler hovers over a K9 within the ancestry tree.
	Step 6: The handler can view the selected K9's ancestry and navigate through the displayed lineage.		
Alternate courses:	**ALT-Step 4:**		

	The system fails to retrieve the K9 ancestry and displays a toaster message: “Failed to retrieve ancestry”. Terminate use case.
Conclusion:	The system generates and displays the family tree of the selected K9.
Post-condition:	The system retrieves and displays the selected K9's ancestry from the K9 table, allowing the handler to view the K9's lineage up to the configured number of generations.
Business rules:	• Only super admin or users given appropriate permissions can view a dog’s ancestry.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 106: View K9 Ancestry - Technical Use Case Narrative

2.9 Rhino Subsystem Use Case Narratives

2.9.1 Add Rhino

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 14 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Add Rhino</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">9.1</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin adds a rhino. The system will display the rhinos and allow the super admin to add a new rhino. The system will display the Add Rhino modal and allow the super admin to enter the new rhino’s details after which the system validates the details entered. The use case ends once the success toaster is displayed and the new rhino has successfully been captured on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to add a rhino.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to add a new rhino.</td><td></td><td>Step 2: The system invokes use case 9.3(View Rhinos).</td></tr>
<tr><td>Step 3: The super admin selects the “Add Rhino” button.</td><td></td><td>Step 4: The system loads and displays the Add Rhino modal using the Angular frontend with the following controls:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			A header section containing: • A heading displaying Create Rhino. A form section containing: • A Name text box for entering the rhino's name. • A Species drop-down list containing the available rhino species. • A Weight numeric input field for entering the rhino's weight (optional). • A Gender drop-down list containing the available gender options. • A Last Seen date picker for selecting the date the rhino was last seen (optional). • A Date of Birth date picker for selecting the rhino's date of birth (optional). • A Province text box for entering the province where

INTEGRATED NGO MANAGEMENT SYSTEM			
			the rhino is located. • A Horn Status drop-down list containing the available horn status options. • An Identification Notes text area for entering additional identification information. • An Image file upload control for selecting an image of the rhino (optional). • A display of the selected image filename when an image has been chosen. An error message section containing: • A validation or processing error message displayed when an error occurs during the rhino creation process. A footer section containing: • A Cancel button to close the Create Rhino modal without creating a rhino.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Create button to proceed with the rhino creation process. The button remains disabled until all required fields have been completed.
	Step 5: The super admin enters the new rhino's details and selects the "Create" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All fields are completed. • The animal's name must not contain numeric characters. • The date of birth, if provided, cannot be a future date. • The last seen date, if provided, cannot be a future date. • The animal weight, if provided, must be greater than 0. • The uploaded image is of type jpeg, png, jpg or webp. [ALT]
			Step 7: The system loads and displays the Add Rhino Confirmation modal using the Angular frontend with the following controls:

INTEGRATED NGO MANAGEMENT SYSTEM			
			A header section containing: • A heading displaying Confirm Create Rhino. A confirmation section containing: • A confirmation message asking the super admin to confirm the creation of the rhino. A footer section containing: • A Cancel button to close the confirmation modal without creating the rhino. • A Confirm button to submit the request to create the rhino.
	Step 8: The super admin selects the "Confirm" button. [ALT]		Step 9: The Angular frontend calls the Rhino service to send a POST request to the C# Web API backend Rhino controller through the CreateRhino endpoint. The backend creates the Rhino in the following entities: Animal Entity: • AnimalId (PK) • AnimalName • AnimalSpecies

INTEGRATED NGO MANAGEMENT SYSTEM			
			• AnimalWeight • AnimalDateOfBirth • AnimalGender • LastSeen • IdentificationNotes • Province • AnimalPhoto • ActiveStatus Rhino Entity: • RhinoId (PK) • AnimalId (FK) • HornStatus [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the rhino. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a rhino was created. AuditLog Entity:

INTEGRATED NGO MANAGEMENT SYSTEM			
			• AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a success toaster informing the admin that the rhino was successfully added.
Alternate courses:	[ALT] Step 5: The super admin selects the "Cancel" button. Use case terminates.		
	[ALT] Step 6: The validation fails. The system displays an error message. Go back to step 5.		
	[ALT] Step 7: The super admin selects the "Cancel" button. Go back to step 5.		
	[ALT] Step 9: The system fails to create the rhino. The system will display an error message based on the status code.		
Conclusion:	The system displays a success toaster informing the super admin that the rhino was added successfully.		
Post-condition:	The rhino is saved on the system.		
Business Rules:	• Super admins or users with the required permissions can add rhinos.		
Implementation constraints and specifications:	• The user requires internet access to access this feature. • Exact locations of rhinos are not saved for security reasons.		
Assumptions:	• None		
Open issues:	• None		

Table 107: Add Rhino - Technical Use Case Narrative

2.9.2 View Rhino

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 14 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Rhino</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">9.2</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin views a rhino’s details. The system will display the rhinos and allow the super admin to filter and search for a rhino. The super admin will then select the rhino to view. The system will retrieve and display the selected rhino’s details. The use case ends once the rhino’s details are displayed.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super Admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to view a rhino.</td></tr>
<tr><td rowspan="5">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to view a rhino.</td><td></td><td>Step 2: The system invokes use case 9.3 (View Rhinos).</td></tr>
<tr><td>Step 3: The super admin applies the required filters to locate the rhino they wish to view. [ALT]</td><td></td><td>Step 4: The Angular frontend filters the retrieved rhinos based on the applied filters. [ALT]</td></tr>
<tr><td>Step 5: The super admin selects the rhino.</td><td></td><td>Step 6: The Angular frontend calls the Rhino service to send a GET request to the C# Web API</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			backend Rhino controller through the GetRhinoById endpoint. The backend retrieves the rhino from the following entities: Rhino Entity: • RhinoId (PK) • AnimalId (FK) • HornStatus • DehornedDate Animal Entity: • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalWeight • AnimalDateOfBirth • AnimalGender • LastSeen • IdentificationNotes • Province • AnimalPhoto • ActiveStatus [ALT]
			Step 6: The system loads and displays the Rhino Details modal using the Angular frontend with the following controls populated with the selected rhino details: A header section containing: • A heading displaying the

INTEGRATED NGO MANAGEMENT SYSTEM			
			selected rhino's name. A details section containing: • A display of the rhino's image. • A display of the rhino's species. • A display of the rhino's gender. • A display of the rhino's weight. • A display of the rhino's location (province). • A display of the date the rhino was last seen. • A display of the rhino's age. • A display of the rhino's horn status. • A display of the rhino's active status. • A display of the dehorned date, if applicable. • A display of the rhino's identification notes. A footer section containing: • An Edit button to open the Update Rhino modal. The

INTEGRATED NGO MANAGEMENT SYSTEM			
			button is disabled when the rhino is inactive. • A Deactivate button to open the Deactivate Rhino confirmation modal. The button is disabled when the rhino is inactive. • A Close button to close the Rhino Details modal.
Alternate courses:	[ALT] Step 3: The super admin does not apply any filters and selects one of the rhinos already displayed. Continue from step 5.		
	[ALT] Step 4: No rhinos match the applied filters, display "No rhinos found" message and go back to step 3.		
	[ALT] Step 6: The system fails to retrieve the selected rhino's details. The system will display an error toaster message based on the status code.		
Conclusion:	The selected rhino's details are displayed.		
Post-condition:	The rhino's details are successfully displayed.		
Business Rules:	• Super admins or users with the required permissions can view a rhino.		
Implementation constraints and specifications:	• The user requires internet access to access this feature.		
Assumptions:	• None		
Open issues:	• None		

Table 108: View Rhino - Technical Use Case Narrative

2.9.3 View Rhinos

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Christopher Bailey		**Date: 14 July 2026** **Version: 1**
USE CASE NAME:	View Rhinos		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	9.3		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	• None		
Description:	This use case describes the event in which a super admin views all rhinos on the system. The admin will navigate to and select the Animals tab and then select the Rhino tab. The system will load and display the Rhino screen. The use case ends once the system displays all the rhinos on the system.		
Pre-condition:	Super admin is logged in.		
Trigger:	Super admin requests to view all rhinos.		
Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: Super admin requests to view all rhinos.		
	Step 2: The super admin navigates to and selects the Animals tab. [ALT]		
	Step 3: The super admin selects the “Rhino” tab.		Step 4: The Angular frontend calls the Projects service to send a GET request to the C# Web API backend Projects controller through the GetProjectLookup endpoint.

INTEGRATED NGO MANAGEMENT SYSTEM			
			The backend retrieves active project details from the following entities: Project Entity: • ProjectId (PK) • ProjectName [ALT]
			Step 5: The Angular frontend calls the Rhino service to send a GET request to the C# Web API backend Rhino controller through the GetAllRhinos endpoint. The backend retrieves the rhinos from the following entities: Rhino Entity: • RhinoId (PK) • AnimalId (FK) • HornStatus Animal Entity: • AnimalId (PK) • AnimalName • AnimalDateOfBirth • AnimalGender • AnimalSpecies • Province • AnimalPhoto • ActiveStatus [ALT]
			Step 6: The Angular frontend calls the Rhino service to retrieve the rhino distribution data from the C# ASP.NET Web API backend through the GetRhinoDistribution endpoint. The backend

INTEGRATED NGO MANAGEMENT SYSTEM			
			retrieves the data from the following entities: Rhino Entity: • RhinoId (PK) • AnimalId (FK) Animal Entity: • AnimalId (PK) • Province • ActiveStatus The backend retrieves active rhinos, groups them by Province, and counts the number of rhinos in each province. [ALT]
			Step 7: The system loads and displays the Rhino screen using the Angular frontend with the following controls: A header section containing: • A heading displaying Rhinos. • An Add Rhino button to open the Create Rhino modal. An interactive map that is implemented using the Leaflet JavaScript mapping library. Rhino location data retrieved from the backend is displayed either as map markers or as a heatmap using the Leaflet.heat plugin, allowing users to visualize both individual rhino locations and areas with

INTEGRATED NGO MANAGEMENT SYSTEM			
			higher rhino concentrations. The map section contains: • An interactive map displaying the geographical distribution of active rhinos. • A Switch to Heatmap/Switch to Markers button to toggle between the marker view and heatmap view of the rhino distribution. A filter section containing: • A Search by Name search box for filtering rhinos by name. • A Project dropdown list containing active projects retrieved from the backend. • A Status drop-down list containing All Status, Active, and Inactive options. • A Species drop-down list containing All Species, Black Rhino, and White Rhino options. • A Gender drop-down list containing All Gender, Male, and Female options. • A Horn Status drop-down list containing

INTEGRATED NGO MANAGEMENT SYSTEM			
			All Horn Status, Has Horn, and Dehorned options. • A Search Location text box for filtering rhinos by province. A rhino display section containing a collection of rhino cards. Each card displays: • The rhino's image. • The rhino's name. • The rhino's gender. • The rhino's species. • The rhino's location. • The rhino's age. • The rhino's horn status. • The rhino's active status. • Each rhino card is selectable and opens the Rhino Details modal for the selected rhino
Alternate courses:	[ALT] Step 2: The super admin is using a mobile device such as a phone. The super admin will select the menu (hamburger) icon and then the Animals option. Continue to step 3.		
	[ALT] Step 4: The system fails to retrieve the lookup data. The system will display an error toaster message based on the status code.		
	[ALT] Step 5: The system fails to retrieve the rhinos. The system will display an error toaster message based on the status code.		
	[ALT] Step 6: The system fails to retrieve the rhino distribution data. The system will display an error toaster message based on the status code.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Conclusion:	The rhinos are successfully displayed.
Post-condition:	All rhinos are displayed.
Business Rules:	• Super admins or users with the required permissions can view rhinos.
Implementation constraints and specifications:	• The user requires internet access to access this feature.
Assumptions:	• None
Open issues:	• None

Table 109: View Rhino - Technical Use Case Narrative

2.9.4 Update Rhino

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Christopher Bailey		**Date: 15 July 2026** **Version: 1**
USE CASE NAME:	Update Rhino		**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	9.4		
PRIORITY:	Medium		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	Super Admin		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	• None		
Description:	This use case describes the event in which a super admin updates a rhino on the system. The system will display the rhino's details and allow the super admin to update the rhino's details on the Update Rhino modal. The system will validate the updated details and ask the super admin for confirmation. Once confirmed the system will save the rhino's updated details. The use case ends once the success toaster is displayed and the rhino's updated details are successfully saved on the system.		
Pre-condition:	Super admin is logged in. Rhino already exists and is active.		
Trigger:	Super admin requests to update a rhino's details.		
Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: Super admin requests to update a rhino's details.		Step 2: The system invokes use case 9.2 (View Rhino).
	Step 3: The super admin selects the "Edit" button.		Step 4: The system loads and displays the Update Rhino modal using the Angular frontend with the following controls populated with the

INTEGRATED NGO MANAGEMENT SYSTEM			
			selected rhino's current details: A modal header containing: • A heading displaying Update Rhino. A modal body containing: • An error message displayed when validation or processing errors occur. • A Name text box for editing the rhino's name. • A Species drop-down list containing Black Rhino and White Rhino options. • A Weight numeric input field for editing the rhino's weight (optional). • A Gender drop-down list containing Male and Female options. • A Last Seen date picker for editing the rhino's last seen date (optional). • A Date of Birth date picker for editing the rhino's date of birth (optional). • An Approximate Location text box. • A Province text box that autocompletes

INTEGRATED NGO MANAGEMENT SYSTEM			
			based on approximate location. • A Horn Status drop-down list containing Has Horn and Dehorned options (If dehorned is selected, the dehorned date is saved). • A Project dropdown. • An Identification Notes text area for editing additional identification information. • A Current Image section displaying the rhino's existing image. • A Replace Image file upload control for selecting a new image (optional). • A display showing the name of the selected replacement image file after it has been chosen. A modal footer containing: • A Cancel button to close the modal without saving changes. • An Update button to validate the entered information and open the Update Rhino Confirmation modal. The Update button

INTEGRATED NGO MANAGEMENT SYSTEM			
			remains disabled until all required fields contain valid values.
	Step 5: The super admin enters the rhino's updated details and selects the "Update" button. [ALT]		Step 6: The system validates the details entered, validating the following: • All required fields are completed. • The rhino's name does not contain numbers. • Last seen and date of birth are not in the future. • The rhino's weight is greater than 0. [ALT]
			Step 7: The system loads and displays the Confirm Rhino Update modal using the Angular frontend with the following controls: A modal header containing: • A heading displaying Confirm Update Rhino. A modal body containing: • A confirmation message asking the super admin whether they wish to update the selected rhino. A set of confirmation actions containing: • A Cancel button to close the confirmation modal

INTEGRATED NGO MANAGEMENT SYSTEM			
			without updating the rhino and return to the Update Rhino modal. • A Confirm button to submit the updated rhino details to the system.
	Step 8: The super admin selects the "Confirm" button. [ALT]		Step 9: The Angular frontend calls the Rhino service to send a PUT request to the C# ASP.NET Web API backend through the UpdateRhino endpoint. The backend updated the rhino in the following entities: Rhino Entity: • RhinoId (PK) • AnimalId (FK) • HornStatus • DehornedDate Animal Entity: • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalWeight • AnimalDateOfBirth • AnimalGender • LastSeen • IdentificationNotes • Province • AnimalPhoto If the rhino's HornStatus changes from Has Horn to Dehorned, the backend sets the DehornedDate to the current date. If a new image

INTEGRATED NGO MANAGEMENT SYSTEM			
			is provided, the backend updates the AnimalPhoto. [ALT]
			Step 10: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful update of the rhino. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a rhino was updated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 11: The system displays a success toaster informing the admin that the rhino was successfully updated.
Alternate courses:	[ALT] Step 5: The super admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 6: The validation fails. The system displays an error message. Go back to step 5.		
	[ALT] Step 8: The super admin selects the “Cancel” button. Go back to step 5.		
	[ALT] Step 9:		

INTEGRATED NGO MANAGEMENT SYSTEM	
	The system fails to update the rhino's details. The system will display an error message based on the status code.
Conclusion:	The success toaster is displayed informing the super admin that the rhino's details were successfully updated.
Post-condition:	Rhino's updated details are successfully saved.
Business Rules:	• Super admins or users with the required permissions can update rhinos.
Implementation constraints and specifications:	• The user requires internet access to access this feature.
Assumptions:	• None
Open issues:	• None

Table 110: Update Rhino - Technical Use Case Narrative

2.9.5 Deactivate Rhino

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 15 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Deactivate Rhino</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">9.5</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin deactivates a rhino on the system. The system will display the rhino’s details and allow the super admin to deactivate the rhino. The system will then ask the super admin for confirmation. Once confirmed the system will deactivate the rhino. The use case ends once the success toaster is displayed and the rhino is successfully deactivated on the system.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">Super admin is logged in.
Rhino already exists and is active.</td></tr>
<tr><td>Trigger:</td><td colspan="3">Super admin requests to deactivate a rhino.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Super admin requests to deactivate a rhino.</td><td></td><td>Step 2: The system invokes use case 9.2 (View Rhino).</td></tr>
<tr><td>Step 3: The super admin selects the “Deactivate” button.</td><td></td><td>Step 4: The system loads and displays the Confirm Rhino Deactivation modal using the Angular frontend with the following controls:</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			A modal header containing: • A heading displaying Confirm Rhino Deactivation. A modal body containing: • An error message displayed if an error occurs during the deactivation process. • A confirmation message asking the super admin whether they wish to deactivate the selected rhino. A set of confirmation actions containing: • A Cancel button to close the confirmation modal without deactivating the rhino and return to the Rhino Details modal. • A Confirm button to deactivate the selected rhino.
	Step 5: The super admin selects the "Confirm" button. [ALT]		Step 6: The Angular frontend calls the Rhino service to send a PUT request to the C# ASP.NET Web API backend through the DeactivateRhino endpoint. The backend updates the rhino's status in the following entity: Rhino Entity: • RhinoId (PK)

INTEGRATED NGO MANAGEMENT SYSTEM			
			• AnimalId (FK) Animal Entity: • AnimalId (PK) • ActiveStatus [ALT]
			Step 7: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful deactivation of the rhino. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a rhino was deactivated. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry
			Step 8: The system displays a success toaster informing the admin that the rhino was successfully deactivated.
Alternate courses:	[ALT] Step 5: The super admin selects the “Cancel” button. Use case terminates.		
	[ALT] Step 6: The system fails to deactivate the rhino. The system will display an error toaster based on the status code.		

INTEGRATED NGO MANAGEMENT SYSTEM	
Conclusion:	The success toaster is displayed informing the super admin that the rhino was successfully deactivated.
Post-condition:	The rhino was successfully deactivated.
Business Rules:	• Super admins or users with the required permissions can deactivate rhinos.
Implementation constraints and specifications:	• The user requires internet access to access this feature.
Assumptions:	• None
Open issues:	• None

Table 111: 9.5 Deactivate Rhino - Technical Use Case Narrative

2.10 Elephant Subsystem Use Case Narratives

2.10.1 Add Elephant

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Add Elephant</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>10.1</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when a Admin adds a new elephant to the system. After navigating to the Animals page and selecting the Elephant tab, the Admin selects “Add Elephant” and fills in a form covering the elephant’s name, species, gender, tusk status, location, and other details, with an optional photo. Once confirmed, the system saves the new elephant across the Animal and Elephant entities and stores the photo if one was provided.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated and on the Animals page.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Add Elephant” from the Elephant grid.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: Admin navigates to the Animals page and selects the “Elephant” tab.</td><td></td><td>Step 2: The system calls “GET api/Elephant” to retrieve all existing elephants by joining the Elephant and Animal entities, then displays them as a grid of cards.</td></tr>
</table>

	Step 3: Admin selects "Add Elephant".		**Step 4:** The system opens a form with fields for Name, Species, Weight (optional), Gender, Last Seen (optional), Date of Birth (optional), Approximate Location, Tusk Status, Identification Notes, and an optional photo upload. The "Create" button is disabled until the form is valid.
	Step 5: Admin completes the form and selects "Create". [ALT]		**Step 6:** The system validates the entered details, for example checking that the name contains no digits and that no dates are set in the future, then displays a confirmation prompt: "Are you sure you want to create this Elephant?"
	Step 7: Admin selects "Confirm". [ALT]		**Step 8:** The ElephantService sends the form data, including the photo if provided, to "POST api/Elephant". If a photo was uploaded, the system stores it on the server. The controller then creates new records in the following entities: **Animal Entity:** • AnimalId (PK) • AnimalName • AnimalSpecies • AnimalWeight • AnimalDateOfBirth • AnimalGender • LastSeen • IdentificationNotes • ApproximateLocation

			• AnimalPhoto • ActiveStatus **Elephant Entity:** • ElephantId (PK) • AnimalId (FK) • TuskStatus
			Step 9: Both records are saved together, with the new elephant's ActiveStatus set to active by default. The API responds with an HTTP 201 Created status, and the frontend refreshes the grid and displays a success notification reading "Successfully created elephant".
Alternate courses:	**[ALT] Step 5:** Admin selects "Cancel". The form closes and resets.		
	[ALT] Step 6: Validation fails. The system displays an inline error message and keeps the form open.		
	[ALT] Step 7: Admin selects "Cancel" on the confirmation prompt. The prompt closes and the form data is kept.		
	[ALT] Step 8a: The API returns an HTTP 400 response due to invalid data. The system displays "Please check the information entered."		
	[ALT] Step 8b: The API returns an HTTP 500 response due to a server error. The system displays "System error, please try again."		
Conclusion:	The use case concludes when the new elephant is successfully saved and appears in the elephant grid.		
Post-condition:	A new record exists in the Animal entity with an active status, along with a linked record in the Elephant entity. The photo is stored on the server if one was provided.		
Business rules:	• Only Admins can add elephants. • Name, species, gender, tusk status, identification notes, and location are required fields.		

Implementation constraints and specifications:	• "POST api/Elephant" accepts the form data, including the optional photo, as a multipart request. • Uploaded photos are limited to JPEG, PNG, and WEBP formats.
Assumptions:	• None.
Open issues:	• None.

Table 112: Add Elephant - Technical Use Case Narrative

2.10.2 View Elephant

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Stephan Bredenhann	**Date:** 13 July 2026 **Version:** 1.0	
Use case name:	View Elephant	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	10.2		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Admin		
Description:	This use case describes the event when an Admin views the full details of a single elephant. From the Elephant grid, the Admin selects a card, and the system retrieves and displays all of that elephant's details in a details window, along with relevant action buttons.		
Pre-condition:	• The Admin is authenticated and the Elephant grid has loaded.		
Trigger:	The Admin selects an elephant card.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Admin navigates to the Animals page, selects the "Elephant" tab, and the grid loads (see Use Case 10.3).		**Step 2:** The grid displays each elephant as a card showing its name, species, location, age, and status.
	Step 3: Admin selects an elephant card. [ALT]		**Step 4:** The system calls "GET api/Elephant/{id}" to retrieve the elephant's details by joining the Elephant and Animal entities. [ALT]

			Step 5: The system opens a details window showing the elephant's photo, Species, Gender, Weight, Location, Last Seen, Age (calculated from date of birth), Tusk Status, Active Status, and Identification Notes, along with "Edit", "Deactivate" (if active) or "Reactivate" (if inactive), and "Close" buttons. [ALT]
	Step 6: Admin reviews the elephant's details.		
Alternate courses:	**[ALT] Step 3:** Admin does not select a card, since the grid currently has no built-in search feature.		
	[ALT] Step 4: The elephant cannot be found. The system does not open the details window and does not display an error message.		
	[ALT] Step 5: A network or server error occurs while retrieving the details. The system does not display an error message to the Admin.		
Conclusion:	The use case concludes when the elephant's details are displayed to the Admin.		
Post-condition:	No data is modified. The elephant's details remain visible until the Admin closes the window.		
Business rules:	• Admins can view the details of any elephant.		
Implementation constraints and specifications:	• "GET api/Elephant/{id}" returns the elephant's details through a read-only query. • The elephant's age is calculated on the frontend from its date of birth.		
Assumptions:	• None.		
Open issues:	• There is no search or filter option in the elephant grid.		

Table 113: View Elephant - Technical Use Case Narrative

2.10.3 View Elephants

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Stephan Bredenhann	**Date:** 13 July 2026 **Version:** 1.0	
Use case name:	View Elephants	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	10.3		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	Admin		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	None		
Description:	This use case describes the event when an Admin views all elephants recorded in the system. After navigating to the Animals page and selecting the Elephant tab, the system retrieves every elephant record and displays them as cards in a grid. Both active and inactive elephants are shown, and the list is not paginated, sorted, or filtered.		
Pre-condition:	• The Admin is authenticated.		
Trigger:	The Admin navigates to the Animals page and selects the "Elephant" tab.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Admin navigates to the Animals page.		**Step 2:** The system displays a tab bar with Rhino, Elephant, and K9 options.
	Step 3: Admin selects the "Elephant" tab.		**Step 4:** The system calls "GET api/Elephant", which retrieves every elephant by joining the Elephant and Animal entities. No filtering,

			sorting, or paging is applied to the query.
			Step 5: For any elephant without a stored photo, the system assigns a default image. The API returns the full list of elephants, and the frontend renders them as a grid of cards showing each elephant's photo, name, species, location, age, and Active/Inactive status.
	Step 6: Admin views the list of elephants.		
Alternate courses:	None.		
Conclusion:	The use case concludes when all elephants, both active and inactive, are displayed in the grid.		
Post-condition:	No data is modified. The elephant grid reflects every elephant record in the system.		
Business rules:	• Admins can view all elephants.		
Implementation constraints and specifications:	• Photo file paths are converted to full URLs by the server before being returned.		
Assumptions:	• None.		
Open issues:	• None.		

Table 114: View Elephants - Technical Use Case Narrative

2.10.4 Update Elephant

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Update Elephant</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>10.4</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin edits an existing elephant’s details. From the elephant details window, the Admin selects “Edit”, updates the pre-filled form, and confirms the change. The system updates the corresponding records across the Animal and Elephant entities and replaces the elephant’s photo if a new one was uploaded.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated and the elephant’s details window is open (see Use Case 10.2).</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Edit” in the elephant details window.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: Admin views an elephant’s details (Use Case 10.2) and selects “Edit”.</td><td></td><td>Step 2: The system closes the details window and opens an update form pre-filled with the elephant’s current Name, Species, Weight, Gender, Last Seen, Date of Birth, Approximate Location, Tusk Status, and</td></tr>
</table>

			Identification Notes, along with the current photo and an option to replace it.
	Step 3: Admin updates the fields and selects "Update". [ALT]		**Step 4:** The system validates the entered details, then displays a confirmation prompt: "Are you sure you want to update this Elephant?"
	Step 5: Admin selects "Confirm". [ALT]		**Step 6:** The ElephantService sends the updated form data to "PUT api/Elephant/{id}". The controller checks that the elephant exists and handles the optional photo upload. [ALT]
			Step 7: Entity Framework Core updates the elephant's TuskStatus on the Elephant entity, and its remaining details on the Animal entity. The photo is only replaced if a new file was uploaded; otherwise, the existing photo is kept. Both updates are saved together. [ALT]
			Step 8: The API responds with an HTTP 204 status. The frontend refreshes the grid, closes the form, and displays a success notification reading

			"Successfully updated elephant".
Alternate courses:	**[ALT] Step 3:** Admin selects "Cancel". The form closes and no changes are made.		
	[ALT] Step 4: Validation fails. The system displays an inline error and keeps the form open.		
	[ALT] Step 5: Admin selects "Cancel" on the confirmation prompt. The prompt closes and the form data is kept.		
	[ALT] Step 6: The elephant cannot be found. The system displays a generic error message.		
	[ALT] Step 7: The API returns an HTTP 500 response due to a server error. The system displays "System error, please try again."		
	[ALT] Step 8: The API returns an HTTP 403 response because the Admin lacks permission. The system displays "You do not have permission to update elephants."		
Conclusion:	The use case concludes when the elephant's updated details are saved and reflected in the grid.		
Post-condition:	The elephant's Animal and Elephant records are updated with the new details. The photo is only changed if a new file was uploaded.		
Business rules:	• Only Admins can update elephant details. • The elephant's Active Status cannot be changed through this use case (see the deactivate and reactivate use cases).		
Implementation constraints and specifications:	• "PUT api/Elephant/{id}" accepts the updated form data, including an optional new photo, as a multipart request. • Field validation is currently performed only on the frontend.		
Assumptions:	• None.		
Open issues:	• None.		

Table 115: Update Elephant - Technical Use Case Narrative

2.10.5 Deactivate Elephant

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Deactivate Elephant</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>10.5</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Admin</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when an Admin deactivates an active elephant using a soft delete. From the elephant details window, the Admin selects “Deactivate” and confirms the action. The system sets the elephant’s ActiveStatus to false on the Animal entity, while keeping the underlying record intact.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The Admin is authenticated, the elephant’s details window is open, and its ActiveStatus is currently true.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The Admin selects “Deactivate” in the elephant details window.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: Admin views an active elephant’s details (Use Case 10.2), where the details window shows a “Deactivate” button.</td><td></td><td></td></tr>
<tr><td>Step 2: Admin selects “Deactivate”.</td><td></td><td>Step 3: The system displays a confirmation prompt: “Are</td></tr>
</table>

			you sure you want to deactivate this Elephant?"
	Step 4: Admin selects "Confirm". [ALT]		**Step 5:** The ElephantService sends a request to "PUT api/Elephant/{id}/deactivate". Entity Framework Core sets the linked Animal record's ActiveStatus to false, without changing any Elephant entity fields. [ALT]
			Step 6: The API responds with an HTTP 204 status. The frontend refreshes the grid, closes the form, and displays a success notification reading "Successfully deactivated elephant".
			Step 7: The deactivated elephant now shows an "Inactive" label in the grid, and its details window shows a "Reactivate" option instead of "Deactivate".
Alternate courses:	**[ALT] Step 4:** Admin selects "Cancel". The prompt closes and no changes are made.		
	[ALT] Step 5a: The elephant cannot be found. The system displays a generic error message.		
	[ALT] Step 5b: The API returns an HTTP 500 response due to a server error. The system does not display an error message to the Admin.		
Conclusion:	The use case concludes when the elephant's ActiveStatus is set to false and reflected in the grid.		
Post-condition:	The elephant's ActiveStatus is false, while all of its other details remain unchanged. The grid shows the elephant as "Inactive", and its details window offers a "Reactivate" option.		
Business rules:	• Only Admins can deactivate elephants. • Deactivation is a soft delete; the elephant record itself is never removed.		

Implementation constraints and specifications:	• "PUT api/Elephant/{id}/deactivate" requires no request body.
Assumptions:	• None.
Open issues:	• None.

Table 116: Deactivate Elephant - Technical Use Case Narrative

2.11 Messages Subsystem Use Case Narratives

2.11.1 Create Message

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 1 May 2026
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">Create Message</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">11.1</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• Group Members (ERA)</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user creates and sends a message on a group. The user will navigate to the messages tab after which the system retrieves all the groups the user is currently in. The user will select a group and the system will retrieve all the messages for that group from the Firebase Firestore database. The user will write their message and send it to the group for other group members to see. The use case ends once the new message is sent and stored in the Firebase Firestore database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User is logged into the system and on their account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to send a message on a group.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to send a message on a group.</td><td></td><td>Step 2: The system invokes use case 11.2 (View Messages)</td></tr>
<tr><td>Step 3: The user writes their message in the message input field and</td><td></td><td>Step 4: The Angular frontend calls the chat service and sends the message details to the</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
	selects to send their message. [ALT]		ASP.NET backend chat controller through the SendMessage endpoint. The backend creates a message object containing the message information and passes it to the FirestoreService, which stores the message in the appropriate group's Firestore message collection. The message is saved with the following details: • senderId • senderName • text • timestamp • groupId [ALT]
			Step 5: All clients subscribed to the group's messages collection receive the new message instantly via the collectionData() observable, and the chat UI updates automatically.
Alternate courses:	**[ALT] Step 3a:** The user decides not to send a message and exits the Messages tab. Use case terminates. **[ALT] Step 3b:** The user selects the microphone button and records an audio message The Angular frontend creates an audio Blob and sends it to the ASP.NET backend.		
	[ALT] Step 4: The Angular frontend calls the chat service and sends the audio Blob to the ASP.NET backend through chat controller through the		

INTEGRATED NGO MANAGEMENT SYSTEM	
	SendVoiceMesssage endpoint. The backend saves the audio file in the wwwroot/voice-messages folder and generates a URL for the saved audio file. The backend then creates a message object containing: • Sender details • Timestamp • Audio URL • Audio duration. The message object is sent to the Firestore service, where the audio URL and message details are stored in the appropriate group's message collection.
Conclusion:	The user's message is saved in the Firebase Firestore database and viewable by other group members.
Post-condition:	The user's message is successfully saved and appears for other users to view.
Business Rules:	• Users can only send messages on groups they are a part of.
Implementation constraints and specifications:	• The user requires internet access to access this feature.
Assumptions:	• None
Open issues:	• None

Table 117: Create Message - Technical Use Case Narrative

2.11.2 View Messages

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 1 May 2026
Version: 1.0</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Message</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">11.2</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">High</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">User</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a user views messages on a group. The user will navigate to the messages tab after which the system retrieves all the groups the user is currently in. The user will select a group and the system will retrieve all the messages for that group from the Firebase Firestore database. The use case ends once all the messages for the selected group are displayed for the user.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">User is logged into the system and on their account.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user requests to view a message on a group.</td></tr>
<tr><td rowspan="4">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user requests to view a message on a group.</td><td></td><td></td></tr>
<tr><td>Step 2: The user navigates to and selects the Messages tab. [ALT]</td><td></td><td>Step 3: The Angular frontend calls the UserMessageGroup service to send a GET request to the C# Web API backend Group controller through the GetUserGroups endpoint. The backend retrieves the user's</td></tr>
</table>

INTEGRATED NGO MANAGEMENT SYSTEM			
			groups from the following entities: Project Entity: • ProjectId (PK) • ProjectName Group Entity: • GroupId (PK) • ProjectId (FK) • GroupName • ActiveStatus UserGroup Entity: • UserId (PK) • GroupId (FK) [ALT]
			Step 4: The system loads and displays the Messages screen using the Angular frontend with the following controls: A screen divided into two main sections, a sidebar and a chat area. The sidebar section: • Displaying a list of projects. • Each project is shown with a project title. • Under each project, a list of groups is displayed. • Each group item includes a chat icon and the group name.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• Clicking on a group opens the corresponding chat. • The selected group is visually highlighted as active. A chat section that changes based on selection: • When no group is selected, an empty state is displayed. • A message indicating No conversation selected. • A prompt instructing the user to select a group from the sidebar to start chatting. • When a group is selected, a chat interface is displayed.
	Step 5: The user clicks the group they wish to view. [ALT]		Step 6: When a group is selected the Angular frontend calls the Chat service to retrieve messages from the Firebase Firestore database.
			Step 7: The Angular frontend displays the messages for the selected group in the chat area: • A header showing the group name. • A messages area displaying a list of messages for the selected group. • A "New Messages" divider is displayed when

INTEGRATED NGO MANAGEMENT SYSTEM			
			applicable to indicate where unread messages begin. • Messages are styled differently depending on whether they are sent by the current user or another user. Each message includes: • The sender's name, shown only for messages from other users. • A message bubble containing the message text. • The time the message was sent. • Audio messages are displayed with an audio player when the message contains a voice recording. The input section at the bottom of the chat provides: • A text input field with a placeholder prompting the user to type a message. • An emoji button that opens an emoji picker, allowing the user to select and insert emojis into their message. • A microphone button that allows the user to record and send an audio message.

INTEGRATED NGO MANAGEMENT SYSTEM			
			• A Send button for sending text messages.
Alternate courses:	**[ALT] Step 2:** The user is using a mobile device such as a phone. The user will select the menu (hamburger) icon and then the Messages option. Continue to step 3.		
	[ALT] Step 3: The system fails to retrieve the user's groups. The system will displays an error toaster based on the status code.		
	[ALT] Step 5: The user does not select a group and exits the messages tab. Use case terminates.		
Conclusion:	The groups messages are displayed.		
Post-condition:	The user can view messages from the selected group.		
Business Rules:	• Users can only view messages in groups they are a part of.		
Implementation constraints and specifications:	• The user requires internet access to access this feature.		
Assumptions:	• None		
Open issues:	• None		

Table 118: View Messages - Technical Use Case Narrative

2.12 Report Subsystem Use Case Narratives

2.12.1 Generate Custom Report

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Stephan Bredenhann	**Date:** 13 July 2026 **Version:** 1.0	
Use case name:	Generate Custom Report	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	12.1		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	User (Authenticated)		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case describes the event when a User builds and generates a custom report. The User selects a domain, chooses which fields to include, optionally applies a filter, and generates an on-screen preview. The report can then be downloaded as a CSV file, and the field selection can be saved as a named configuration for later reuse.		
Pre-condition:	• The User is authenticated and the selected domain contains existing records.		
Trigger:	The User navigates to the Reports page and selects a domain to begin building a report.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: User navigates to the Reports page.		**Step 2:** The system calls “GET api/reports/domains”, which returns the list of reportable domains and the fields available for each one. The domains populate the selection dropdown. [ALT]

	Step 3: User selects a domain from the dropdown.		**Step 4:** The system displays the available fields for that domain as a checkbox grid, with every field selected by default, and loads the user's saved configurations for the domain into the sidebar.
	Step 5: User deselects any unwanted fields and, if required, chooses a filter field and enters a filter value.		**Step 6:** The system records the field selection and filter. No request is made at this point.
	Step 7: User selects "Generate Preview".		**Step 8:** The system checks that a domain is selected and at least one field is chosen, then calls "POST api/reports/generate" with the domain, the selected fields, and any filter. [ALT]
			Step 9: The system retrieves the records for the selected domain, returns only the requested fields, and applies the filter if one was supplied. The results are displayed in a scrollable preview table using the field display names as column headings.
	Step 10: User selects "Download CSV".		**Step 11:** The system calls "POST api/reports/generate/download", which regenerates the report and returns it as a CSV file named after the domain and the current date and time. The browser saves the file and the system displays a success notification. [ALT]
	Step 12: User selects "Save Current Configuration".		**Step 13:** The system opens a window containing a name field, a summary of the selected domain and field count, and "Save" and "Cancel" buttons. The "Save"

			button remains disabled until a name is entered.
	Step 14: User enters a configuration name and selects "Save". [ALT]		**Step 15:** The system calls "POST api/reports/configurations", which inserts a new record into the ReportConfiguration entity holding the name, domain, and field selection against the user. The window closes, the saved configurations sidebar refreshes, and a success notification is displayed. [ALT]
Alternate courses:	**[ALT] Step 2:** The domains cannot be retrieved. The system displays "Failed to load report domains" and the dropdown remains empty.		
	[ALT] Step 8: The report cannot be generated, either because the domain is not recognised or because the database is unavailable. The system displays "Failed to generate report preview".		
	[ALT] Step 11: The report cannot be downloaded. The system displays "Failed to download report" and no file is saved.		
	[ALT] Step 14: The configuration name is left blank. The system displays "Please enter a configuration name" and no request is made.		
	[ALT] Step 15: The configuration cannot be saved. The system displays "Failed to save configuration" and the window remains open.		
Conclusion:	The use case concludes when the report preview is displayed to the User and, if selected, the report is downloaded or the field selection is saved as a named configuration.		
Post-condition:	The report preview is displayed. If the report was downloaded, a CSV file is saved to the User's device. If a configuration was saved, a new record exists in the ReportConfiguration entity holding the field selection.		
Business rules:	• Only authenticated users can generate reports. • At least one field must be selected before a report can be generated. • A configuration must be given a name before it can be saved.		

Implementation constraints and specifications:	• "POST api/reports/generate" accepts the domain, the selected fields, and an optional filter, and returns only the requested fields. • The CSV file is encoded so that it opens correctly in Microsoft Excel. • The preview table scrolls horizontally so that wide reports do not break the page layout.
Assumptions:	• Each reportable domain has been configured with the fields available to it. • The User's browser supports downloading files generated by the page.
Open issues:	• None.

Table 119: Generate Custom Report - Technical Use Case Narrative

2.12.2 View Custom Report

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Custom Report</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>12.2</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Administrator</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when a User views previously saved report configurations. The User can browse all saved configurations, narrow them by domain, and review the name, domain, selected fields, and dates recorded against each one. A configuration can also be loaded into the Report Builder to display its report preview. Only configurations belonging to the User are shown, and no data is created, updated, or deleted.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The User is authenticated and has at least one saved report configuration.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The User navigates to the Saved Configurations page or opens the saved configurations sidebar on the Report Builder page.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: User navigates to the Saved Configurations page.</td><td></td><td>Step 2: The system calls “GET api/reports/configurations”, which retrieves the configurations belonging to the User, ordered by the date they were last updated. [ALT]</td></tr>
</table>

			Step 3: The system displays each configuration as a card showing its name, domain, the fields it includes, and the dates it was created and last updated, together with "Load", "Edit", and "Delete" buttons. [ALT]
	Step 4: User selects a domain from the filter dropdown.		**Step 5:** The system narrows the list to configurations belonging to the selected domain. Selecting "All Domains" restores the full list.
	Step 6: User selects "Load" on a configuration card.		**Step 7:** The system calls "GET api/reports/configurations/{id}", which retrieves the configuration only if it belongs to the User. The system opens the Report Builder page with the saved domain and field selection restored. [ALT]
			Step 8: The system generates the report preview from the restored field selection, as described in Use Case 12.1, and displays the results in the preview table.
Alternate courses:	**[ALT] Step 2:** The User is not authenticated or the session has expired. The system redirects the User to the login page.		
	[ALT] Step 3: The User has no saved configurations. The system displays a message inviting the User to create a report.		
	[ALT] Step 7a: The configuration cannot be found because it was deleted or belongs to another user. The system displays "Failed to load configuration".		
	[ALT] Step 7b: The configuration refers to a domain that no longer exists. The system displays "Domain not found for this configuration" and no report is generated.		

Conclusion:	The use case concludes when the saved configurations are displayed to the User, with the selected configuration loaded into the Report Builder if requested.
Post-condition:	No data is modified. If a configuration was loaded, the Report Builder displays the report preview generated from its field selection.
Business rules:	• Only authenticated users can view report configurations. • Configurations that have been removed are excluded from the list. • Configurations are listed with the most recently updated first.
Implementation constraints and specifications:	• Domain and field labels are resolved from the stored report definitions, so no further request is needed to display them.
Assumptions:	• The reportable domains and their fields remain consistent with the entities they describe. • A configuration saved against a domain remains valid while that domain exists.
Open issues:	• None.

Table 120: View Custom Report - Technical Use Case Narrative

2.12.3 Update Custom Report

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Stephan Bredenhann</td><td colspan="2">Date: 13 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Update Custom Report</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>12.3</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">Administrator</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event when a User updates a saved report configuration. From the Saved Configurations page, the User renames a configuration, and the system updates the record only after confirming the configuration belongs to the User. A User who instead wants to change which fields a report includes loads the configuration into the Report Builder, adjusts the field selection, and saves it as a new configuration.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The User is authenticated and has at least one saved report configuration.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The User selects “Edit” on a configuration card on the Saved Configurations page.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: User navigates to the Saved Configurations page.</td><td></td><td>Step 2: The system displays the User’s saved configurations as cards, each showing the configuration name, domain, and selected fields, with “Load”, “Edit”, and “Delete” buttons.</td></tr>
</table>

	Step 3: User selects "Edit" on a configuration card.		**Step 4:** The system opens an edit window containing a name field pre-filled with the current configuration name, together with "Save" and "Cancel" buttons. The "Save" button remains disabled while the name is blank.
	Step 5: User changes the configuration name and selects "Save". [ALT]		**Step 6:** The system calls "PUT api/reports/configurations/{id}", which retrieves the configuration only if it belongs to the User and updates the name and the last updated date. The domain and field selection are left unchanged. [ALT]
			Step 7: The system closes the edit window, refreshes the configurations list with the new name, and displays a success notification.
	Step 8: User selects "Load" on a configuration to change the fields it includes.		**Step 9:** The system restores the saved domain and field selection in the Report Builder and generates the report preview, as described in Use Case 12.2.
	Step 10: User adds or removes fields and selects "Save Current Configuration".		**Step 11:** The system updates the preview and the selected field count as each field is changed, then saves the revised selection as a new configuration, as described in Use Case 12.1. [ALT]
Alternate courses:	**[ALT] Step 5:** User selects "Cancel" or closes the edit window. No changes are made and no request is sent.		
	[ALT] Step 6a: The configuration cannot be found because it was removed or belongs to another user. The system displays "Configuration not found" and the edit window remains open.		
	[ALT] Step 6b:		

	The User is not authenticated or the session has expired. The system redirects the User to the login page.
	[ALT] Step 6c: The configuration cannot be updated. The system displays "Failed to update configuration" and the edit window remains open.
	[ALT] Step 11: The name entered matches an existing configuration. The system saves it regardless, so two configurations with the same name can exist for the same domain.
Conclusion:	The use case concludes when the configuration is renamed and the new name appears in the list, or when a revised field selection is saved as a new configuration.
Post-condition:	The selected configuration has an updated name and last updated date. If the field selection was changed instead, a new configuration exists holding the revised fields.
Business rules:	• Only authenticated users can update report configurations. • The last updated date is set every time a configuration is changed. • Renaming a configuration does not change the fields it includes.
Implementation constraints and specifications:	• "PUT api/reports/configurations/{id}" updates the name and the last updated date only; the domain cannot be changed once a configuration has been saved.
Assumptions:	• The User can identify which configuration to update from its name, domain, and listed fields. • Configuration names do not need to be unique within a domain.
Open issues:	• None.

Table 121: Update Custom Report - Technical Use Case Narrative

2.12.4 Remove Custom Report

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Stephan Bredenhann	**Date:** 13 July 2026 **Version:** 1.0	
Use case name:	Remove Custom Report	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	12.4		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	User (Authenticated)		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	Administrator		
Description:	This use case describes the event when a User removes a saved report configuration. The configuration is marked as inactive rather than being deleted from the database, so the record is retained for audit purposes while no longer appearing to the User. Removal can be started from the Saved Configurations page or from the Report Builder sidebar, and both paths require confirmation before the configuration is removed.		
Pre-condition:	• The User is authenticated and has at least one saved report configuration.		
Trigger:	The User selects “Delete” on a configuration card on the Saved Configurations page, or on a configuration in the Report Builder sidebar.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: User navigates to the Saved Configurations page.		**Step 2:** The system displays the User’s saved configurations as cards, each showing the configuration name, domain, and selected fields, with “Load”, “Edit”, and “Delete” buttons.
	Step 3: User selects “Delete” on a		**Step 4:** The system displays a confirmation prompt reading

	configuration card. [ALT]		"Are you sure you want to delete the configuration?", naming the selected configuration, with "Delete" and "Cancel" buttons.
	Step 5: User selects "Delete" to confirm. [ALT]		**Step 6:** The system calls "DELETE api/reports/configurations/{id}", which retrieves the configuration only if it is active and belongs to the User. [ALT]
			Step 7: The system marks the configuration as inactive rather than removing the record, leaving all other details intact. [ALT]
			Step 8: The system refreshes the configurations list, which no longer includes the removed configuration, and displays a success notification.
Alternate courses:	**[ALT] Step 3:** User selects "Delete" on a configuration in the Report Builder sidebar instead. The system displays a confirmation prompt and once confirmed, follows the same removal process from Step 6.		
	[ALT] Step 5: User selects "Cancel" or closes the confirmation prompt. The configuration is not removed and no request is sent.		
	[ALT] Step 6a: The configuration cannot be found because it was already removed or belongs to another user. The system displays "Configuration not found" and refreshes the list.		
	[ALT] Step 6b: The User is not authenticated or the session has expired. The system redirects the User to the login page.		
	[ALT] Step 7: The configuration cannot be updated. The system displays "Failed to delete configuration" and the configuration remains in the list.		
Conclusion:	The use case concludes when the configuration is marked as inactive, no longer appears in the configurations list, and a success notification is displayed.		

Post-condition:	The selected configuration is marked as inactive and is no longer returned to the User. The record is retained with its name, domain, field selection, and dates intact.
Business rules:	• Only authenticated users can remove report configurations. • Removal is not destructive; the record is retained and marked as inactive. • Only active configurations can be removed. • The User must confirm the removal before it is carried out.
Implementation constraints and specifications:	• "DELETE api/reports/configurations/{id}" marks the configuration as inactive rather than deleting the record, and no other details are changed. • A configuration that has already been removed returns the same result as one that does not exist.
Assumptions:	• The confirmation prompt is sufficient to prevent accidental removals.
Open issues:	• None.

Table 122: Remove Custom Report - Technical Use Case Narrative

2.12.5 View Bike Impact Report

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Alexander Steven	**Date:** 24 July 2026 **Version:** 1.0	
USE CASE NAME:	View Bike Impact Report	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	12.5		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	User		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes a user viewing the Bike Impact Report from the Admin section of the system. The user opens the report page, where the system loads a year-to-date view showing bicycle deployment, beneficiary attendance and cost figures for all bike projects. The user can narrow or widen the period using preset ranges or specific dates and can download the report as a PDF or CSV file. Every view or export is written to the audit log.		
Pre-condition:	• The user is logged in and holds the Reports.BikeImpact.View permission issues by Use Case 2.16 Update Role Permission.		
Trigger:	The user selects Bike Impact Report from the Admin menu.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: User selects Bike Impact Report from the Admin menu.		**Step 2:** The system checks that the user is logged in and holds the required permission, then loads the report page with the Year-to-date period applied by default.

			Step 3: The system requests the report from the API, which reads the attendance, inventory and expense records for the period and returns the finished report.
			Step 4: The page displays the key figures at the top, followed by the report's sections covering project summaries, attendance, bicycle deployment and cost.
	Step 5: User changes the reporting period, either by choosing a different preset or entering two dates and selects Generate report. [ALT]		**Step 6:** The system repeats the same checks and rebuilds the report for the new period.
	Step 7: User selects Download PDF.		**Step 8:** The system rebuilds the report, produces a PDF on the server, and the browser saves the file. The system records the export in the audit log. **AuditLogs Entity:** • UserId (FK) • Action • EntityType • EntityId • Timestamp
Alternate courses:	**[ALT] Step 1a:** User is not logged in. The system redirects to the login page; a direct request to the API returns an unauthorised response.		
	[ALT] Step 1b:		

	User is logged in but does not hold the Reports.BikeImpact.View permission. The system redirects to a not-authorised page; a direct request returns a forbidden response.
	[ALT] Step 3a: No bike project falls inside the selected period. The report still returns successfully, showing a single message that the period was empty rather than leaving any section blank.
	[ALT] Step 3b: A section has no rows for the period even though the rest of the report does. That section shows its own short message instead of an empty table.
	[ALT] Step 5a: User enters a start date later than the end date. The Generate report button is disabled and an inline message explains why.
	[ALT] Step 5b: User selects a period longer than five years. The system rejects the request and asks for a shorter range.
Conclusion:	The use case concludes when the user has viewed the report on screen or has downloaded it as a PDF or CSV file.
Post-condition:	The report has been displayed to the user and, if requested, saved as a file. No attendance, inventory or expense record has been changed; the only change is the new entry in the audit log.
Business Rules:	• A bike project is included in the report if any part of its timeline falls inside the selected period, not only if it started there. • Attendance records are matched to present, present with bicycle, or absent; anything that cannot be matched is counted separately rather than guessed at.
Implementation constraints and specifications:	• The report page is served from /reports/bike-impact and is protected by a route guard and a permission guard on the Angular side.
Assumptions:	• The user has been assigned the correct role and permissions by an administrator before requesting this report.
Open issues:	• None.

Table 123: View Bike Impact Report - Technical Use Case Narrative

2.12.6 View Budget Report

INTEGRATED NGO MANAGEMENT SYSTEM			
AUTHOR:	Alex Steven	**Date:** 24 July 2026 **Version:** 1.0	
USE CASE NAME:	View Budget Report	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
USE CASE ID:	12.6		
PRIORITY:	High		
SOURCE:	EPI-USE		
Primary Business Actor (PBA):	User		
Primary System Actor (PSA):	• None		
Other Participating Actors:	• None		
Other Interested Stakeholders:	None		
Description:	This use case describes a user viewing the Budget Report from the Admin section of the system. The user opens the report page, where the system loads a year-to-date view comparing each project's approved budget against actual and committed spend. The user can narrow or widen the period using preset ranges or specific dates and can download the report as a PDF or CSV file. Every view or export is written to the audit log.		
Pre-condition:	• The user is logged in and holds the Reports.Budget.View permission.		
Trigger:	The user selects Budget Report from the Admin menu.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: User selects Budget Report from the Admin menu.		**Step 2:** The system checks that the user is logged in and holds the required permission, then loads the report page with the Year-to-date period applied by default.
			Step 3: The system requests the report from the API, which

			reads the expenses, purchase orders, projects and inventory for the period, works out how much of each budget has been committed or spent, and returns the finished report.
			Step 4: The page displays the key figures at the top, followed by the report's sections covering budget against actual spend, spend by category, spend by supplier and any budget exceptions.
	Step 5: User changes the reporting period and selects Generate report. [ALT]		**Step 6:** The system repeats the same checks and rebuilds the report for the new period.
	Step 7: User selects Download PDF.		**Step 8:** The system rebuilds the report, produces a PDF on the server, and the browser saves the file. The system records the export in the audit log. **AuditLogs Entity:** • UserId (FK) • Action • EntityType • EntityId • Timestamp
Alternate courses:	**[ALT] Step 1a:** User is not logged in. The system redirects to the login page; a direct request to the API returns an unauthorised response.		
	[ALT] Step 1b:		

	User is logged in but does not hold the Reports.Budget.View permission. The system redirects to a not-authorised page; a direct request returns a forbidden response.
	[ALT] Step 3a: A project has no approved budget recorded. It is shown with a dash instead of a budget figure and listed separately, rather than being treated as a budget of zero.
	[ALT] Step 3b: A project's spend and commitments exceed its budget. It is marked as over budget and listed in the exceptions section.
	[ALT] Step 5a: User enters a start date later than the end date. The Generate report button is disabled and an inline message explains why.
	[ALT] Step 5b: User selects a period longer than five years. The system rejects the request and asks for a shorter range.
Conclusion:	The use case concludes when the user has viewed the report on screen or has downloaded it as a PDF or CSV file.
Post-condition:	The report has been displayed to the user and, if requested, saved as a file. No financial record has been changed; the only change is the new entry in the audit log.
Business Rules:	• A purchase order counts as committed spend only until an expense settles it, so nothing is counted twice.
Implementation constraints and specifications:	• The report page is served from /reports/budget and is protected by a route guard and a permission guard on the Angular side.
Assumptions:	• The user has been assigned the correct role and permissions by an administrator before requesting this report.
Open issues:	• None.

Table 124: View Budget Report - Technical Use Case Narrative

2.12.7 View Rhino Report

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Amahle Khoza</td><td>Date: 24/07/2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td colspan="2">View Rhino Report</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">12.7</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Administrator</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how an authorized user views a pre-structured Rhino Report within a selected date range. The user specifies the filter criteria, and the system fetches, calculates, and presents aggregated rhino operational, tracking, and health metrics.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be logged into the system and possess appropriate permissions.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user wants to generate and view a Rhino Report for a specific date range.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td>Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td></td><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user navigates to the Reports module and selects the "Rhino Report" option.</td><td></td><td>Step 2: The system loads and displays the Rhino Report filter page, containing date range selection inputs (Start Date and End Date) and a "Generate Report" button. The Generate Report button remains disabled until valid date values are selected.</td></tr>
<tr><td></td><td>Step 3: The user selects the Start</td><td></td><td>Step 4: The system validates the selected date parameters using</td></tr>
</table>

	Date and End Date parameters and clicks the "Generate Report" button. [ALT]		Angular frontend validation before submitting the query. The system checks that: • The Start Date field is not null or empty. • The End Date field is not null or empty. • The Start Date is less than or equal to the End Date. [ALT]
			Step 5: The system routes the query through the Angular frontend and Report Service, which dispatches an HTTP GET request via the Angular HTTP Client to the corresponding Web API endpoint in Visual Studio (ReportsController – GetRhinoReport).
			Step 6: The API endpoint validates user authentication/RBAC claims and passes the date parameters to the Report Repository/Service. An optimized, read-only (.AsNoTracking()) Entity Framework Core query is executed against Microsoft SQL Server to fetch matching records (Timestamp >= StartDate && Timestamp <= EndDate). The following entities are involved in the process: **Rhino Sighting / Observation Entity:** • RhinoId (PK) • TagNumber • SpeciesType • RecordedDate • LocationCoordinates • HealthStatus

			[ALT]
			Step 7: The backend calculates aggregate metrics, maps the results into a RhinoReportResultDto payload, and returns an HTTP 200 OK response to the Angular frontend.
			Step 8: The system records the view action in the AuditLog table for auditing and compliance tracking. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry ("Viewed Rhino Report for period X to Y")
			Step 9: The system renders the generated Rhino Report on the screen, displaying metric summary cards (e.g., total observations, active alerts) along with a structured data table of observation records.
Alternate courses:	**ALT-Step 3:** The user changes their mind and selects the "Clear Filters" or "Cancel" button. The form resets. Terminate use case.		
	ALT-Step 4: The Start Date is greater than the End Date. The system displays an inline validation error: *"Start date cannot be greater than end date."*		
	ALT-Step 6: The system fails to retrieve data or encounters a server timeout, returning a toaster message: *"Failed to generate report. Please try again later."* Terminate use case.		
Conclusion:	The requested Rhino Report for the specified date range is successfully generated, displayed on the screen, and logged in the audit trail.		
Post-condition:	The system presents the structured Rhino Report to the user without altering underlying observation records.		
Business rules:	• Only users with appropriate role-based permissions (e.g., Admin, Manager, Ranger Lead) can view Rhino Reports.		

	• Users must be authenticated and maintain an active session.
Implementation constraints and specifications:	• Pre-structured data projection utilizing read-only (.AsNoTracking()) EF Core query execution for optimal query performance.
Assumptions:	• Rhino observation and health tracking data are correctly populated in the database.
Open issues:	• None

Table 125: View Rhino Report - Technical Use Case Narrative

2.12.8 View Elephant Report

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Amahle Khoza</td><td>Date: 24/07/2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td colspan="2">View Elephant Report</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">12.8</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">Administrator</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes how an authorized user generates and views an Elephant Report for a specified date range. The system processes parameter inputs, executes data aggregation queries, and presents structured telemetry, herd movement, and activity data.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user must be logged into the system and possess appropriate permissions.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user wants to view an Elephant Report for a designated timeframe.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user navigates to the Reports module and selects the "Elephant Report" option.</td><td></td><td>Step 2: The system loads and displays the Elephant Report parameter view; rendering Start Date and End Date input controls and a "Generate Report" action button. The Generate Report button remains disabled until both date inputs are populated.</td></tr>
<tr><td>Step 3: The user inputs the desired Start Date and</td><td></td><td>Step 4: The system validates the date parameters via Angular frontend form controls prior to</td></tr>
</table>

	End Date values and selects the "Generate Report" button. [ALT]		API submission. Validation criteria include: • Start Date field is not null or empty. • End Date field is not null or empty. • Start Date is less than or equal to End Date. [ALT]
			Step 5: The system routes the query request via the Angular frontend and Report Service through the Angular HTTP Client to the corresponding Web API endpoint in Visual Studio (ElephantReportsController – GetElephantReport).
			Step 6: The API endpoint enforces JWT security and role claims, delegating the request to the Elephant Report Service. An optimized, read-only (.AsNoTracking()) LINQ query runs against Microsoft SQL Server to aggregate matching telemetry and event logs (LoggedAt >= StartDate && LoggedAt <= EndDate). The following entities are involved in the process: **Elephant Observation / Telemetry Entity:** • ElephantId (PK) • IdentifierCode • HerdIdentifier • LoggedAt • ObservedSector • ActivityType [ALT]

			Step 7: The server projects the aggregated datasets into an ElephantReportResultDto structure and transmits an HTTP 200 OK response back to the client UI.
			Step 8: The backend writes a transaction log entry into the AuditLog table for historical tracking and regulatory auditing. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • LogEntry ("Viewed Elephant Report for period X to Y")
			Step 9: The system renders the populated Elephant Report layout, presenting high-level herd metric summary panels alongside structured detailed event tables.
Alternate courses:	**ALT-Step 3:** The user selects "Cancel" or clears the form controls. The report generation process terminates.		
	ALT-Step 4: Date range validation fails (Start Date > End Date). The system presents an error alert: *"Invalid temporal range. Start date must precede End date."* Go back to Step 3.		
	ALT-Step 6: Database query processing fails. The system throws an error toaster: *"Failed to fetch elephant report metrics."* Terminate use case.		
Conclusion:	The Elephant Report for the requested date range is successfully generated, displayed in the web application, and recorded in system audit logs.		
Post-condition:	The Elephant Report is displayed on-screen in a read-only view. No underlying telemetry data is modified.		

Business rules:	• Only authenticated users with role-based permissions (e.g., Admin, Manager, Conservationist) can view Elephant Reports. • Session bearer token must be valid.
Implementation constraints and specifications:	• High-performance aggregate data retrieval using indexed temporal fields and non-tracking EF Core queries.
Assumptions:	• Elephant movement and herd telemetry records are populated and accessible in SQL Server.
Open issues:	• None

Table 126: View Elephant Report - Technical Use Case Narrative

2.12.9 View K9 Report

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Christopher Bailey</td><td>Date: 25 July 2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View K9 Report</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">12.9</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Medium</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes the event in which a super admin views a K9 report. The super admin will navigate to the Admin tab in the navbar and will then select Reports and then the K9 Report option. The system will retrieve the K9 report’s details from the backend API and display the report to the super admin. The super admin will then have the option to save the report as a pdf or export CSV. The use case ends once the super admin saves the report.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• Super admin is logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to view the K9 report.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin requests to view the K9 report.</td><td></td><td></td></tr>
<tr><td>Step 2: The admin navigates to the Admin tab and selects</td><td></td><td>Step 3: The Angular frontend calls the StructuredReportService service to send a GET request to the C# Web API backend StructuredReports controller through the GetK9</td></tr>
</table>

	Reports then K9 Report. [ALT]		endpoint. The backend retrieves the report details from the following entities: K9 Entity: • K9Id (PK) • AnimalId (FK) Animal Entity: • AnimalId (PK) • AnimalName • AnimalSpecies (Breed) • AnimalGender • AnimalDateOfBirth • ActiveStatus K9Role Entity: • K9RoleId (PK) • RoleName K9K9Role Entity: • K9Id (PK, FK) • K9RoleId (PK, FK) K9User Entity: • AssignmentId (PK) • K9Id (FK) • UserId (FK) • StartDate • EndDate User Entity: • UserId (PK) • UserName • UserSurname AnimalProject Entity: • AnimalId (PK, FK) • ProjectId (PK, FK) Project Entity:

			• ProjectId (PK) • ProjectName • StartDate • EndDate • IsActive Inventory Entity: • InventoryItemId (PK) • ProjectId (FK) • InventoryTypeId (FK) • WarehouseId (FK) • ModelName • Value InventoryType Entity: • InventoryTypeId (PK) • Name Warehouse Entity: • WarehouseId (PK) • Name [ALT]
			Step 4: The system will then record the transaction in the AuditLog table to maintain a history of actions performed within the system. This is handled in the backend after the successful creation of the report. The system will store information such as the authenticated user performing the action, the timestamp of the operation, and a descriptive log entry indicating that a K9 report has been created. AuditLog Entity: • AuditLogID (PK) • UserID (FK) • Timestamp

			• LogEntry
			Step 5: The system loads and displays the K9 Operational Report screen using the Angular frontend with the following controls: A header section containing: • A heading displaying **K9 Operational Report**. • A description displaying **K9 register, handler assignments and operational activity**. • A reporting period displaying the selected date range. • A Generated-On value displaying the report generation date and time. • A Generated By value displaying the user who generated the report. A report filter section containing: • A Start Date field allowing the user to select the beginning of the reporting period. • An End Date field allowing the user to select the end of the reporting period. • Preset date range buttons including: • This Month. • This Quarter. • Year to Date. • Last 12 Months.

			• A Generate Report button to regenerate the report for the selected period. • A Download PDF button to export the report as a PDF document. • A Download CSV button to export the report as a CSV file. A KPI summary section containing: • A **Dogs on Register** KPI displaying the total number of active K9s. • A **Retired Dogs** KPI displaying the number of inactive K9s. • An **Active Handlers** KPI displaying the number of handlers currently assigned to operational K9s. • An **Active Assignments** KPI displaying the number of current handler assignments. • **An Active K9 Register section containing a table with the following columns:** • Name displaying the K9's name. • Breed / Species displaying the K9 breed or species. • Gender displaying the K9's gender. • Age displaying the calculated age of the K9.

			• Current Handler displaying the currently assigned handler. • Operational Roles displaying the operational role or roles assigned to the K9. A Current Handler Assignments section containing a table with the following columns: • Dog displaying the K9 name. • Handler displaying the assigned handler. • Assignment Started displaying the assignment start date. • Days Assigned displaying the number of days the current assignment has been active. • A fallback message displaying **"No current handler assignments."** when no current assignments exist. A Completed Handler Assignments section containing a table with the following columns: • Dog displaying the K9 name. • Handler displaying the assigned handler. • Assignment Started displaying the assignment start date. • Assignment Ended displaying the assignment end date.

			• Days Assigned displaying the duration of the completed assignment. • A fallback message displaying **"No handler assignments were completed during this reporting period."** when no completed assignments exist. An Operational Roles section containing a table with the following columns: • Dog displaying the K9 name. • Current Handler displaying the assigned handler. • Operational Roles displaying the operational role or roles assigned to the K9. A Role Distribution section containing: • A breakdown chart displaying the number of K9s assigned to each operational role. • A fallback message displaying **"No K9 roles have been assigned."** when no role assignments exist. A Project Assignments section containing a table with the following columns: • Project displaying the project name. • K9 displaying the assigned K9.

			• Current Handler displaying the currently assigned handler. • Project Status displaying whether the project is Active or Closed. • Project Started displaying the project start date. • Project End displaying the project end date. • A fallback message displaying **"No K9 project assignments."** when no project assignments exist. A K9s by Project section containing: • A breakdown chart displaying the number of K9s assigned to each project. • A fallback message displaying **"No project assignments."** when no project assignments exist. An Inventory Supporting K9 Operations section containing a table with the following columns: • Project displaying the project name. • Inventory Item displaying the inventory item name. • Inventory Type displaying the inventory category. • Warehouse displaying the warehouse in which the item is stored.

			• Estimated Value displaying the recorded monetary value of the inventory item. • A fallback message displaying **"No inventory allocated to K9 projects."** when no inventory has been allocated. An Inventory by Type section containing: • A breakdown chart displaying the number of inventory items grouped by inventory type. • A fallback message displaying **"No inventory has been allocated."** when no inventory records exist. An Observations section containing: • A narrative list highlighting operational findings identified from the report data, including K9s without handlers, K9s without operational roles, K9s without projects, projects without allocated inventory, handlers responsible for multiple K9s, K9s with multiple active handler assignments, and the total recorded value of inventory allocated to K9 projects. • A fallback message displaying **"No operational or data quality issues were detected for K9 deployments during the selected reporting**

			period." when no issues are identified.
	Step 6: The super admin selects "Download PDF" button. [ALT]		**Step 7:** The Angular frontend calls the StructuredReportService service to send a GET request to the C# Web API backend StructuredReports controller through the GetK9Pdf endpoint. The backend validates the selected reporting period and retrieves the required K9, animal, handler, assignment, project, inventory, inventory type and warehouse data from the database. The generated data is then rendered using the **QuestPDF** library to compose the report into a structured PDF document. The backend records the PDF export in the audit log and returns the generated PDF file as a file response. The Angular frontend receives the PDF as a **Blob** and automatically downloads it to the user's device using the filename format: **k9-report_YYYY-MM-DD_YYYY-MM-DD.pdf**
Alternate courses:	**[ALT] Step 2:** The admin is using a mobile device such as a phone. The admin will select the menu (hamburger) icon and then the Admin option. Continue to step 3.		
	[ALT] Step 3: The system fails to retrieve the report data. The system will display an error message.		
	[ALT] Step 6a: The super admin does not save the pdf, just views it.		
	[ALT] Step 6b: The super admin selects "Export CSV". The Angular frontend calls the StructuredReportService service to send a GET request to the C# Web API backend StructuredReports controller through the GetK9Csv endpoint.		

	The backend validates the selected reporting period and retrieves the required K9, animal, handler, assignment, project, inventory, inventory type, and warehouse data from the database. The backend iterates through each report section and converts the report tables into **Comma-Separated Values (CSV)** format. The backend records the CSV export in the audit log and returns the generated CSV file as a file response. The Angular frontend receives the CSV file as a **Blob** and automatically downloads it to the user's device using the filename format: **k9-report_YYYY-MM-DD_YYYY-MM-DD.csv**
Conclusion:	The K9 report has been saved and downloaded.
Post-condition:	K9 report is saved.
Business Rules:	• Super admins can view K9 report.
Implementation constraints and specifications:	• None
Assumptions:	• None
Open issues:	• None

Table 127: View K9 Report - Technical Use Case Narrative

2.12.10 View Machine Learning Dashboard

INTEGRATED NGO MANAGEMENT SYSTEM		
AUTHOR:	Stephan Bredenhann	**Date:** 26 July 2026 **Version:** 1.0
USE CASE NAME:	Backup System	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	12.10 View Machine Learning	
PRIORITY:	Medium	
SOURCE:	EPI-USE	
Primary Business Actor (PBA):	Super Admin	
Primary System Actor (PSA):	• None	
Other Participating Actors:	• None	
Other Interested Stakeholders:	None	
Description:	This use case describes how a Super Admin views the predictive analytics ML Dashboard, covering budget forecasting, team size recommendation, risk scoring, and inventory demand across all ongoing projects. The dashboard is never pre-computed: every time it is opened, the ".NET 8" API (PredictionsController, PredictionService, TrainingDataService) assembles a fresh feature matrix from live operational data and dispatches it to a separate Python FastAPI sidecar (ml-service/app.py), which fits scikit-learn pipelines and returns predictions with cross-validated diagnostics. The use case concludes with the Super Admin reviewing project risk cards, drilling into a single project's detail, and checking predicted stock shortfalls.	

Pre-condition:	• The Super Admin must be authenticated and mapped to an active user record. PredictionAccessService.EnsureCanViewPredictionsAsync() enforces authentication only ;there is no additional role restriction beyond the domain's own access rules.		
Trigger:	The Super Admin selects “Machine Learning Dashboard” from the main navigation menu.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: Super Admin selects “Machine Learning Dashboard” from the main navigation menu.		**Step 2:** The Angular frontend sends an authenticated GET request to “/api/predictions/dashboard”. PredictionsController first calls PredictionAccessService.EnsureCanViewPredictionsAsync() to confirm the caller is an authenticated, active user.
			Step 3: TrainingDataService runs 8 set-based queries against Projects, ProjectDeadlines, Locations, AnimalProjects, ProjectVendors, Permits, TeamRoles, UserProjects/UserProjectRoles/ProjectTeamRoleMembers, Groups/UserGroups, Inventory/InventoryTypes, ProcurementLines, UserSkills, ProjectFeedbacks, and Expenses, assembling the feature matrix (ProjectFeatureCatalogue, 25 base + 4 progress features) for every ongoing project alongside the completed-project training set.
	Step 4: Super Admin selects “Create Backup Now” and confirms “Create a full database		**Step 4:** PredictionService builds four concurrent FitPredictRequest payloads (budget, team, risk, stock) and posts each to the Python sidecar's “POST /fit-predict” endpoint at the configured ModelService:BaseUrl (default

	backup now?". [ALT]		"http://127.0.0.1:8000", 5-second timeout). [ALT]
			Step 5: For each task with at least 8 completed training rows, the FastAPI service runs its scikit-learn pipeline (SmoothedTargetEncoder → SelectKBest → StandardScaler → RidgeCV / LogisticRegression), fits the model, cross-validates it (leave-one-out for the regression tasks, stratified k-fold for risk classification), and returns predictions plus a diagnostics block (r2Loo, mae, accuracyCv, selected features, coefficients).
			Step 6: PredictionService assembles the MlDashboardDto, combining each ongoing project's budget/team/risk predictions and PredictionMetaDto confidence data with the stock array (30/60/90-day demand versus stock-on-hand) and a modelInfo summary, then returns "200 OK". **MlDashboardDto:** • generatedAtUtc • projects[] • stock[] • modelInfo
	Step 7: Super Admin reviews the dashboard: project cards sorted by risk (highest first) showing budget delta, team recommendation, and a		**Step 9:** BackupService.RestoreAsync() looks up the backup, validates the file exists inside the configured backup directory, snapshots the current disk ".bak" files, acquires the semaphore, and calls MaintenanceState.Enter() ;from this point every request outside "/api/backups/*" receives "503". [ALT]

	colour-banded risk percentage; a stock shortfall panel; and a model info footer reporting service health and sample sizes.		
	Step 8: Super Admin selects a project card to drill into its detail page.		**Step 9:** The frontend navigates to the project detail route and sends an authenticated GET request to "/api/predictions/projects/{projectId}", which returns that project's budget, team, and risk predictions (including the top contributing risk factors) for display. [ALT]
	Step 10: Super Admin returns to the dashboard and selects "View Stock Predictions".		**Step 11:** The frontend calls "GET /api/predictions/stock", and the stock shortfall panel renders predicted demand against available and allocated stock for each inventory type across the 30, 60, and 90-day horizons.
	Step 12: Super Admin selects "Refresh Dashboard".		**Step 13:** The frontend re-issues "GET /api/predictions/dashboard", causing all four models to be refit against the now-current live data (Steps 2–6 repeat).
	Step 14: Super Admin selects "Delete" on a backup entry and confirms "Permanently delete this backup?".		

Alternate courses:	**[ALT] Step 3**: Fewer than 8 completed projects exist for a given task. PredictionService skips the model fit and applies the task's baseline directly (median spend by category for budget, median team size of well-rated projects for team, a burn-vs-schedule heuristic for risk, average monthly purchasing for stock), setting basis: "Baseline" and a note such as "Needs 8 completed projects to fit a model; N available".
	[ALT] Step 4: The Python sidecar is unreachable (connection refused or the 5-second timeout elapses). PredictionService catches the failure and falls back to the same baseline computations for every affected task; the dashboard still renders in full, never returning an error to the Super Admin.
	[ALT] Step 5: The sidecar receives fewer than its own MIN_ROWS = 4 completed rows (a backstop behind the .NET layer's 8-row gate) and returns a validation error with reason="insufficient rows"; PredictionService falls back to baseline for that task.
	[ALT] Step 9: “GET /api/predictions/projects/{projectId}” is called for a project that is no longer ongoing or does not exist. The controller returns “404 Not Found” and the frontend shows an error toast, returning the Super Admin to the dashboard.
Conclusion:	The use case concludes when the Super Admin has reviewed the current predictive analytics for ongoing projects, whether summarized on the dashboard, drilled into a single project's detail, or checked as a stock demand forecast.
Post-condition:	No database state changes because of this use case ;it is read-only. The dashboard reflects a model fit computed from live operational data now it was requested; reopening or refreshing it repeats the fit rather than reusing a cached result.
Business Rules:	• A model is fit only when at least 8 completed projects are available for that task; below this threshold the system always falls back to a baseline rather than failing. • Confidence is reported as "High" only when sample size is at least 20 and R^2 is at least 0.5, "Medium" or "Low" otherwise.
Implementation constraints and specifications:	• The Python sidecar is a stateless FastAPI service (“ml-service/app.py”) reached via a typed HttpClient registered against ModelServiceOptions (BaseUrl, TimeoutSeconds); it is started independently of the “.NET” API with “uvicorn app:app”.

Assumptions:	• Historical completed-project data is assumed to be representative of current operating conditions; predictions naturally carry lower confidence for project categories with few historical examples.
Open issues:	• None

Table 128: View Machine Learning Dashboard - Technical Use Case Narrative

2.13 System Subsystem Use Case Narratives

2.13.1 Backup System

INTEGRATED NGO MANAGEMENT SYSTEM		
AUTHOR:	Stephan Bredenhann	**Date:** 26 July 2026 **Version:** 1.0
USE CASE NAME:	Backup System	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	13.1	
PRIORITY:	High	
SOURCE:	EPI-USE	
Primary Business Actor (PBA):	Super Admin	
Primary System Actor (PSA):	• None	
Other Participating Actors:	• None	
Other Interested Stakeholders:	None	
Description:	This use case describes how a Super Admin manages full database backups and restores through the Backup Management page, and how the system additionally performs backups automatically on a schedule. Manual and scheduled backups both use SQL Server's native BACKUP DATABASE command, staging the resulting ".bak" file through a SQL Server service-account-writable directory before the ".NET 8" API copies it into its own "secure-backups/" store and records metadata in the Backups table. A restore enters maintenance mode (returning "503" on all other requests), restores the database in single-user mode, then reconciles backup metadata from the ".bak" files that survive the rollback. The use case concludes with the Super Admin having created, restored, deleted, or rescheduled backups, or a scheduled backup completing unattended.	
Pre-condition:	• The Super Admin must be authenticated (enforced by the global fallback authorization policy in Program.cs). For a restore, at least one backup record with a corresponding ".bak" file must exist.	
Trigger:	The Super Admin navigates to the Backup Management page, or one-minute elapses on BackupSchedulerHostedService's polling timer.	

Typical Course of Events:	Actor Action	System Response	
		Manual Action	Automated Action
	Step 1: Super Admin navigates to the Backup Management page from the admin menu.		**Step 2:** The frontend sends three authenticated GET requests in parallel: "/api/backups", "/api/backups/status", and "/api/backups/schedule". "GET /api/backups" additionally checks whether the Backups table is empty while ".bak" files exist on disk (for example, following a prior restore) and, if so, auto-rebuilds the metadata rows from disk before returning the list.
			Step 3: The list, status ({backupInProgress, maintenanceMode}), and schedule ({isEnabled, dayOfWeek, timeOfDay, lastSuccessfulRunAt}) responses populate the backup list table, the status indicator, and the schedule panel respectively.
	Step 4: Super Admin selects "Create Backup Now" and confirms "Create a full database backup now?". [ALT]		**Step 5:** BackupService.CreateAsync() acquires a SemaphoreSlim(1,1), sets backupInProgress to true, computes the filename {DatabaseName}_{yyyyMMdd_HHmmss}_{Guid:N}.bak, and resolves the SQL Server staging directory from BackupSettings:SqlServerDirectory or SERVERPROPERTY('InstanceDefaultBackupPath').
			Step 6: The service executes BACKUP DATABASE ... TO DISK = @stagingPath WITH FORMAT, INIT, NAME, SKIP, NOREWIND, NOUNLOAD, STATS = 10, then copies the resulting file into "secure-backups/" and deletes the staging copy. [ALT]
			Step 7: A metadata row is inserted into the Backups table, PruneAutomaticBackupsAsync() removes any automatic backups beyond MaxAutomaticBackups (default 5), an AuditLogs entry (Action="Create") is written, and the semaphore is released; the API returns the new backup and the frontend refreshes the list. **Backups Entity:** • BackupId (PK) • Date • FilePath

			• Size • IsAutomatic • CreatedByUserId (FK)
	Step 8: Super Admin selects "Restore" on an existing backup entry and confirms "Restore will overwrite the current database and sign everyone out. Continue?". [ALT]		**Step 9:** BackupService.RestoreAsync() looks up the backup, validates the file exists inside the configured backup directory, snapshots the current disk ".bak" files, acquires the semaphore, and calls MaintenanceState.Enter() ;from this point every request outside "/api/backups/*" receives "503". [ALT]
			Step 10: The service copies the file to the SQL staging directory if needed, connects to the master database, and executes ALTER DATABASE SET SINGLE_USER WITH ROLLBACK IMMEDIATE, then RESTORE DATABASE ... WITH REPLACE, then ALTER DATABASE SET MULTI_USER. [ALT]
			Step 11: In a fresh scope, the service re-scans disk for ".bak" files, clears and re-registers the Backups table from those files, re-seeds the BackupScheduleSettings singleton if missing from the restored snapshot, writes an AuditLogs entry (Action="Restore"), exits maintenance mode, and releases the semaphore. The API responds "Database restored successfully. Please sign in again."
	Step 12: Super Admin opens the Schedule panel, adjusts the enabled toggle, day of week, and time of day,		**Step 13:** "PUT /api/backups/schedule" validates that dayOfWeek is 0–6 and timeOfDay is a valid HH:mm value, updates the BackupScheduleSettings singleton row (Id = 1), and returns the updated schedule. [ALT]

	then selects Save.		
	Step 14: Super Admin selects "Delete" on a backup entry and confirms "Permanently delete this backup?".		**Step 15:** "DELETE /api/backups/{id}" removes both the ".bak" file from disk and its metadata row, writes an AuditLogs entry (Action="Delete"), and returns "Backup deleted.". [ALT]
			Step 16: Independently of any admin action, BackupSchedulerHostedService polls every minute. When the schedule is enabled, the current day matches DayOfWeek, the current time falls within the 5-minute window after TimeOfDay, and no successful run has been recorded for that window, it calls CreateAsync(isAutomatic: true, userId: null) ;repeating Steps 5–7 with IsAutomatic = true and an audit Action of "AutoCreate" ;then updates LastSuccessfulRunAt to the current UTC time on success. [ALT]
Alternate courses:	**[ALT] Step 4:** A backup is already in progress. "POST /api/backups" returns "409 Conflict"; the frontend shows "A backup is already running. Please wait." and disables the Create/Delete/Restore buttons until status polling reports backupInProgress = false.		
	[ALT] Step 6: BACKUP DATABASE fails (for example, disk full or the staging directory is not writable). The service returns "500 Internal Server Error"; the semaphore is still released so the system is not left locked, and the frontend shows a toast with the error message.		
	[ALT] Step 8: Super Admin selects Download instead of Restore. "GET /api/backups/{id}/download" validates the file path is inside the configured backup directory via EnsurePathIsInsideBackupDirectory(), then streams the ".bak" file as application/octet-stream; "404" if the backup record or file is missing.		
	[ALT] Step 9: The requested backup's file cannot be found on disk, or the record does not exist. RestoreAsync() returns "400 Bad Request" before entering maintenance mode, so no other users are affected.		

	[ALT] Step 10: RESTORE DATABASE fails mid-operation (for example, a corrupted file). The service returns "500 Internal Server Error"; the frontend shows an error toast, and maintenance mode is still exited so the system remains usable while the Super Admin investigates.
	[ALT] Step 13: dayOfWeek is outside 0–6 or timeOfDay is not a valid HH:mm value. The endpoint returns "400 Bad Request" and the frontend shows an inline validation message without saving.
	[ALT] Step 15: The backup ID does not exist. "DELETE /api/backups/{id}" returns "404 Not Found"; the frontend shows "Backup file not found on disk."
	[ALT] Step 16: The scheduler tick fires while the semaphore is already held by a manual backup or restore in progress. TryRunScheduledBackupAsync() cannot acquire the lock immediately, skips this tick, and returns false without retry until the next minute's poll.
Conclusion:	The use case concludes when the Super Admin has successfully created, restored, deleted, or rescheduled a database backup, or when a scheduled backup completes automatically in the background without any admin action.
Post-condition:	The Backups table and "secure backups/" directory accurately reflect the current set of available backups, BackupScheduleSettings reflects the active schedule, and every operation performed is recorded in AuditLogs.
Business Rules:	• All backup endpoints require authentication; there is no additional role check beyond the global fallback authorization policy. • Only one backup or restore operation may run at a time, enforced process-wide by a SemaphoreSlim(1,1). • Automatic backups beyond MaxAutomaticBackups (default 5) are pruned after every run; manual backups are never pruned automatically. • During a restore, every request outside "/api/backups/*" receives "503" until maintenance mode is exited.
Implementation constraints and specifications:	• SQL Server writes backups to a staging directory that must be writable by the SQL Server service account, distinct from the app's own "secure-backups/" directory (BackupSettings:SqlServerDirectory, auto-detected via SERVERPROPERTY('InstanceDefaultBackupPath') when unset).
Assumptions:	• The SQL Server service account is assumed to have write access to the configured staging directory; if not,

	PromoteStagingFileToAppStorageAsync() throws a descriptive configuration error rather than failing silently.
Open issues:	• None

Table 129: Backup System - Technical Use Case Narrative

2.13.2 Restore System

INTEGRATED NGO MANAGEMENT SYSTEM		
AUTHOR:	Stephan Bredenhann	**Date:** 26 July 2026 **Version:** 1.0
USE CASE NAME:	Restore System	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓
USE CASE ID:	13.2	
PRIORITY:	High	
SOURCE:	EPI-USE	
Primary Business Actor (PBA):	Super Admin	
Primary System Actor (PSA):	• BackupService, MaintenanceMiddleware	
Other Participating Actors:	• SQL Server	
Other Interested Stakeholders:	All authenticated users	
Description:	This use case describes how a Super Admin replaces the entire database with a prior state captured in a ".bak" file. Restore is implemented inside the same BackupService.cs as backup creation (BackupService.RestoreAsync()), but is a distinct, strictly destructive, manually-triggered operation: it can never be scheduled or automated. While it runs, the system enters a process-wide maintenance mode in which MaintenanceMiddleware ;mounted ahead of authentication and routing ;returns "503" for every request, with no carve-out for backup endpoints. After the RESTORE DATABASE command completes, the system reconciles backup metadata from whichever ".bak" files survived the rollback, since the restored database snapshot almost never contains the metadata row for the backup that was just taken.	
Pre-condition:	• The Super Admin must be authenticated. At least one Backups record must exist whose ".bak" file is present and readable inside the configured backup directory, and no other backup or restore operation may currently hold the system-wide lock.	
Trigger:	The Super Admin selects "Restore" on a backup row and confirms the destructive warning dialog.	
	Actor Action	**System Response**

		Manual Action	Automated Action
Typical Course of Events:	**Step 1:** Super Admin selects "Restore" on a backup entry on the Backup Management page.		**Step 2:** The frontend displays a confirmation dialog: "This will overwrite the entire database with the backup from {date}. All current data will be replaced. Everyone (including you) will be signed out. The system will be unavailable during the restore. This action cannot be undone.". [ALT]
	Step 3: Super Admin confirms by selecting "Restore" in the dialog.		**Step 4:** The frontend sends "POST /api/backups/{id}/restore". BackupsController.Restore() calls BackupService.RestoreAsync(id, userId), which looks up the backup by BackupId (AsNoTracking), confirms the ".bak" file exists on disk, and validates the path with EnsurePathIsInsideBackupDirectory(). [ALT]
			Step 5: SnapshotDiskBackupFilesOnDisk() enumerates every ".bak" file in "secure-backups/", recording each file's path, size, and last-write time. This snapshot is essential: the Backups table itself will be rolled back by the restore, and a backup's own metadata row is inserted only after BACKUP DATABASE completes, so it is almost never present in the restored snapshot.
			Step 6: The service acquires the SemaphoreSlim(1,1) lock [ALT] and calls MaintenanceState.Enter(reason), setting the process-wide IsInMaintenance flag. From this instant, MaintenanceMiddleware ;mounted

			at the very top of the pipeline, ahead of authentication and routing ;returns “503 Service Unavailable” with the reason message for every subsequent request, with no exception path for “/api/backups/*”. [ALT]
			Step 7: If the app's “secure-backups/” directory differs from the SQL Server staging directory (resolved from BackupSettings:SqlServerDirectory or SERVERPROPERTY('InstanceDefaultBackupPath')), the “.bak” file is copied into the staging directory, overwriting any existing file there; the staging copy is tracked for cleanup regardless of outcome. [ALT]
			Step 8: The service connects to the master database (InitialCatalog overridden to “master”, since the target database cannot be restored while any connection to it remains open) and executes, with no command timeout, ALTER DATABASE SET SINGLE_USER WITH ROLLBACK IMMEDIATE, then RESTORE DATABASE FROM DISK = @path WITH REPLACE, STATS = 10, then ALTER DATABASE SET MULTI_USER. [ALT]
			Step 9: ReconcileBackupsFromDiskAsync() runs inside a freshly created IServiceScope, since the original request-scoped DbContext is dead after SINGLE_USER killed its connection. It re-scans “secure-backups/”, removes all existing Backups rows, and re-inserts one row per “.bak” file found (IsAutomatic = false and CreatedByUserId = null in every case, since the original creator

			identity cannot be recovered), then re-seeds the BackupScheduleSettings singleton if the restored snapshot predates that table. **BackupScheduleSettings Entity:** • BackupScheduleSettingsId (PK) • IsEnabled • DayOfWeek • TimeOfDay
			Step 10: An AuditLogs entry is written (Action = "Restore", EntityId = the restored filename), any staging copy is deleted, MaintenanceState.Exit() clears the flag, and the semaphore is released. The API returns "200 OK" with "Database restored successfully. Please sign in again."
	Step 11: Per the response message, the frontend clears the stored authentication token and redirects the Super Admin ;and every other signed-in user whose next request lands after maintenance mode exits ;to the login page.		
Alternate courses:	**[ALT] Step 2:** Super Admin selects "Cancel" instead of "Restore" in the confirmation dialog. The dialog closes and no request is sent; the database is left untouched.		
	[ALT] Step 4: The backup ID does not exist, or its recorded ".bak" file cannot be found on disk. RestoreAsync() throws an InvalidOperationException, which the controller returns as "400 Bad Request" with detail "Backup not found." or "Backup file is missing from disk."; maintenance mode is never entered.		
	[ALT] Step 6:		

	Another backup or restore is already in progress and the semaphore cannot be acquired immediately. RestoreAsync() throws InvalidOperationException("A backup or restore is already in progress."), returned as “400 Bad Request”; the in-flight operation is left undisturbed.
	[ALT] Step 7: The SQL Server staging directory is not writable, or a WSL/Windows path mismatch prevents the copy. The staging helper throws a descriptive configuration error; the resulting “500 Internal Server Error” surfaces “Restore failed: {exception message}”, and maintenance mode and the semaphore are still released via the finally block.
	[ALT] Step 8: RESTORE DATABASE fails (a corrupted “.bak” file, insufficient disk space, or a permissions error). The exception surfaces as “500 Internal Server Error” with “Restore failed: {exception message}”; the finally block still exits maintenance mode and releases the semaphore so the system remains available for the Super Admin to retry.
	[ALT] Step 6 (during maintenance): any user ;including the Super Admin's own subsequent requests ;hitting any endpoint, including “GET /api/backups/status”, during the maintenance window receives “503” with the configured reason; the frontend's HTTP interceptor detects this and displays a blocking banner across the entire application until maintenance ends.
Conclusion:	The use case concludes when the target database has been fully replaced by the selected “.bak” file's contents, the Backups table and BackupScheduleSettings have been reconciled from disk, and the system has exited maintenance mode and is accepting requests again.
Post-condition:	The application database matches the state captured in the restored “.bak” file. The Backups table lists exactly the “.bak” files present on disk at reconciliation time, each as a manual entry with no recoverable creator; BackupScheduleSettings exists (re-seeded with defaults if it was missing); an AuditLogs “Restore” entry has been written; and MaintenanceState.IsInMaintenance is false again. Any user whose session depends on data that no longer matches the restored snapshot must sign in again.
Business Rules:	• Only one backup or restore operation may run at a time system-wide, enforced by the same SemaphoreSlim(1,1) used for backup creation.
Implementation constraints and specifications:	• None.

Assumptions:	• The SQL Server service account is assumed able to read the staged ".bak" file and to have permission to execute ALTER DATABASE and RESTORE DATABASE against the target database.
Open issues:	• None.

Table 130: Restore System - Technical Use Case Narrative

2.13.3 View Audit Log

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>AUTHOR:</td><td colspan="2">Nicholas Bailey</td><td>Date: 25/07/2026
Version: 1</td></tr>
<tr><td>USE CASE NAME:</td><td colspan="2">View Audit Log</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>USE CASE ID:</td><td colspan="2">13.3</td></tr>
<tr><td>PRIORITY:</td><td colspan="2">Low</td></tr>
<tr><td>SOURCE:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary Business Actor (PBA):</td><td colspan="3">Super Admin</td></tr>
<tr><td>Primary System Actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other Participating Actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other Interested Stakeholders:</td><td colspan="3">• None</td></tr>
<tr><td>Description:</td><td colspan="3">This use case describes when the super admin views audit logs recorded within the system. The system retrieves and displays audit log records in a table, including details such as the user who performed the action, the action performed, the affected entity, and additional information about the activity. The system also provides audit statistics, searching, and pagination functionality to assist the super admin in monitoring system activity. The use case ends when the super admin can view and review recorded system actions and their details.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The super admin must be logged into the system.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The super admin requests to view the full audit log.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The super admin wants to view all audit logs.</td><td></td><td></td></tr>
<tr><td>Step 2: The super admin hovers over the "Admin" navigation tab and</td><td></td><td>Step 3: The system sends the request through the Angular frontend and AuditLogService to the relevant HTTP GET API endpoints in Visual Studio</td></tr>
</table>

	selects the "Audit Logs" option. [ALT]		(AuditLogsController – GetAll and AuditLogsController – GetStats). The GetAll endpoint retrieves audit log records from the AuditLogs table in the database through the AuditLogRepository, while the GetStats endpoint retrieves audit activity statistics used for the summary section. **AuditLog Entity:** • AuditLogID (PK) • UserID (FK) • Timestamp • Action • EntityType • EntityID • Details
			Step 4: The system then displays the Audit Logs page using the Angular frontend, showing the retrieved audit log records and statistics returned from the API. • Page header section: ○ Page title ("Audit Logs") ○ Description text ("View all recorded system actions performed by users.")

			• Statistics section: ○ Total Logs ○ Last 24 Hours ○ Last 7 Days • Search section: ○ Search input field for filtering audit logs. ○ Search button to retrieve matching records. ○ Displays the total number of logs found. • Table display section: ○ Audit logs are displayed in a table. ○ The table includes the following columns: ▪ Date ▪ User ▪ Action ▪ Entity ▪ Details • Table behaviour: ○ Each row represents a

			single audit log record. ○ Records are displayed in chronological order, with the most recent log entries appearing first. ○ Pagination controls allow the super admin to navigate between pages of audit records. [ALT]
	Step 5: The super admin can view, search, and monitor all recorded system actions performed by users.		
Alternate courses:	**ALT-Step 2:** User is on mobile and selects the menu icon. Go to step 3.		
	ALT-Step 4: The system fails to load the audit log data and displays an error message.		
Conclusion:	The system has successfully retrieved a list of all the audit logs within the system.		
Post-condition:	The system has successfully retrieved a list of all the audit logs within the system.		
Business Rules:	• Only super admins or users given appropriate permissions can view audit logs. • Ability to Backup Database.		
Implementation constraints and specifications:	• Must be user-friendly and easy to use.		

Assumptions:	• None
Open issues:	• None

Table 131: View Audit Log - Technical Use Case Narrative

2.14 Budget Subsystem Use Case Narratives

2.14.1 View Project Budget

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Project Budget</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>14.1</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user with the appropriate permissions should be able to view a project budget. The system will allow the user to select and view a budget’s details including expenses. The user clicks on a specific project from the main budget dashboard. The Angular frontend sends a request to the .NET 8 API to fetch the financial data. The backend queries the "Project" and "Expense" database tables, calculates the total amount spent, and groups the data so it can be drawn as charts. The frontend then displays a clean dashboard showing a proportional breakdown (pie chart), a cumulative timeline (line chart), and a full transaction history table.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is logged in, has the correct permissions to view financial data, and is on the main Budget screen.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks on a specific project card or table row on the Budget dashboard.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks on a specific</td><td></td><td>Step 2: The Angular frontend navigates to the detailed view and sends a GET</td></tr>
</table>

	project to view its budget details.		api/Expenses/project/{projectId} request to the backend.
			Step 3: The API controller receives the request and uses Entity Framework Core to verify the "Project" exists in the database. [ALT]
			Step 4: The system queries the "Expense" table for all records linked to that specific project ID, ordered by the date they were incurred. [ALT]
			Step 5: The backend calculates the total spent, groups the expenses by category for the pie chart, and calls a helper function to generate the cumulative timeline data for the line chart.
			Step 6: The API packages this formatted data into a "ProjectBudgetDetailDto" and returns an HTTP 200 OK success response.
			Step 7: The Angular frontend receives the data and renders the visual elements: a "Total Logged" number, a "Proportional Breakdown" pie chart, an "Expense Timeline" line graph, and the "Transaction History" table.
Alternate courses:	**[ALT] Step 3:** The requested project ID cannot be found in the database. The API returns an HTTP 404 Not Found response. The frontend redirects the user back to the main dashboard and displays a red error toast: "Project budget not found."		
	[ALT] Step 4: The database connection drops while trying to retrieve the expenses. The API returns an HTTP 500 Internal Server Error. The frontend leaves the		

	charts empty and displays a red error toast: "Failed to load financial data."
Conclusion:	The user successfully views a detailed, visual breakdown of the project's financial expenses.
Post-condition:	The financial data is displayed on the screen. No database records are modified during this view process.
Business rules:	Financial data is privileged; only users assigned to the project or with administrative roles may view it.
Implementation constraints and specifications:	• The backend controller formats the chart data (labels, values, and hex colours) before sending it to the frontend, which keeps the Angular code lightweight.
Assumptions:	• The frontend charting library can handle dynamically plotting the data slices sent by the backend.
Open issues:	• While the user interface shows a "Transaction History" table, the ability to edit or delete existing expenses directly from this table may need to be mapped out in future use cases.

Table 132: View Project Budget - Technical Use Case Narrative

2.14.2 View Project Budgets

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>View Project Budgets</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>14.2</td></tr>
<tr><td>Priority:</td><td>High</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">System Admin (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user with the appropriate permissions should be able to view all project budgets. The system will retrieve all project budgets on the system and display them to the user. The user navigates to the "Budget" tab, which triggers the Angular frontend to request the dashboard data from the .NET 8 API. The backend retrieves the user's assigned projects, fetches all related expenses across the company, and calculates global totals. It packages the proportions, cumulative timelines, and table data into a "DashboardSummaryDto", which is then dynamically rendered on the frontend as interactive charts and a comprehensive project list.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is logged into the system and has the correct role permissions to view the financial dashboard.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "Budget" tab in the main navigation menu.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td></td><td>Step 1: The user clicks the "Budget" tab on the navigation bar.</td><td></td><td>Step 2: The Angular frontend detects the navigation and sends a GET api/Expenses/dashboard/{userId} request to the backend.</td></tr>
</table>

			Step 3: The API controller receives the request and uses Entity Framework Core to fetch the user's project assignments from the "UserProjects" table. [ALT]
			Step 4: The system queries the "Expenses" table to get all financial logs, and the "Projects" table to get active project details.
			Step 5: The backend calculates the grand total, groups the expenses into a "BigChart" (pie chart) and a "BigTimeline" (line chart), and formats the data for the recent projects cards and the "TableProjects" list.
			Step 6: The API returns an HTTP 200 OK response containing the fully populated "DashboardSummaryDto" JSON object.
			Step 7: The Angular frontend parses the data and renders the overall Budget & Expenses dashboard, including the interactive charts and the "All Active Projects" data table.
Alternate courses:	**[ALT] Step 3:** The database connection fails while trying to fetch the user's projects or expenses. The API returns an HTTP 500 Internal Server Error. The frontend stops rendering the charts and displays a red error toast: "Failed to load financial data."		
Conclusion:	The user is presented with a comprehensive, visual overview of all project budgets they are authorized to see.		
Post-condition:	The aggregated financial data is displayed. No database records are altered.		
Business rules:	Financial data is privileged, so the dashboard must only display project data that the user's role explicitly permits them to view.		

Implementation constraints and specifications:	• The backend controller assigns distinct colors from a predefined "Palette" array to ensure the pie charts and line charts remain visually consistent across the dashboard.
Assumptions:	• The frontend charting libraries can dynamically bind to the nested arrays (like List<ChartSliceDto>) provided in the response payload.
Open issues:	• Fetching all expenses across all projects simultaneously might cause performance bottlenecks as the database grows over the years

Table 133: View Project Budgets - Technical Use Case Narrative

2.14.3 Log An Expense

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td colspan="2">Alexander Steven</td><td>Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td colspan="2">Log An Expense</td><td rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td colspan="2">14.3</td></tr>
<tr><td>Priority:</td><td colspan="2">High</td></tr>
<tr><td>Source:</td><td colspan="2">EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">User (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user with the appropriate role can log a new expense for a specific project. The user clicks the "+ Log Expense" button on the project's budget dashboard, opening a modal to capture the amount (ZAR), category, and an optional receipt file. The Angular frontend packages this into an "AddExpenseDto" and sends a POST request to the .NET 8 API. The backend verifies that the user is an active member of the project, creates the "Expense" record (storing a snapshot of the user's name at that exact moment), and saves it to the database.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is authenticated and navigating the Budget dashboard for a specific project.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "+ Log Expense" button.</td></tr>
<tr><td rowspan="3">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "+ Log Expense" button.</td><td></td><td>Step 2: The Angular frontend opens the "Log New Expense" modal.</td></tr>
<tr><td>Step 3: The user enters the amount, selects a category, optionally uploads a receipt, and clicks</td><td></td><td>Step 4: The Angular frontend packages the "AddExpenseDto" and sends a POST</td></tr>
</table>

	"Save Expense". [ALT]		api/Expenses request to the backend.
			Step 5: The API controller receives the request and queries the "Users" and "UserProjects" database tables to ensure the user exists and is assigned to the project. [ALT]
			Step 6: The system initializes a new "Expense" entity, setting the "LoggedByName" snapshot and mapping the DTO fields.
			Step 7: The system adds the record and calls SaveChangesAsync().[ALT]
			Step 8: The API returns an HTTP 200 OK success response.
			Step 9: The Angular frontend closes the modal, refreshes the budget charts to reflect the new total, and displays a success toast.
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel". The modal closes and no data are submitted.		
	[ALT] Step 5: The user is not assigned to the project. The API returns an HTTP 403 Forbidden response. The frontend displays a red error toast: "You are not authorized to log expenses for this project."		
	[ALT] Step 7: The database connection fails. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast.		
Conclusion:	The financial transaction is successfully logged and instantly reflected on the project's financial dashboard.		
Post-condition:	A new record is inserted into the "Expenses" table.		

Business rules:	Users can only log expenses for projects they are actively participating in.
Implementation constraints and specifications:	• The backend captures the "LoggedByName" string permanently so historical records remain accurate even if a user deletes their account later.
Assumptions:	• The frontend handles the file upload for the receipt beforehand and passes the resulting "ReceiptUrl" in the DTO.
Open issues:	• None

Table 134: Log An Expense – Technical Use Case Narrative

2.14.4 Update An Expense

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Alexander Steven	**Date:** 15 July 2026 **Version:** 1.0	
Use case name:	Update An Expense	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	14.4		
Priority:	High		
Source:	EPI-USE		
Primary business actor (PBA):	User (Authenticated)		
Primary system actor (PSA):	• None		
Other participating actors:	• None		
Other interested stakeholders:	None		
Description:	A user with the appropriate permissions should be able to edit the details of an expense. The system will display the expense details in a modal, allowing the user to modify the amount, category, or update the receipt URL. Once the user clicks "Save", the Angular frontend validates the input and sends a PUT request to the .NET 8 API. The backend queries Entity Framework Core to verify the expense exists, applies the modifications to the database record, and responds with a success status.		
Pre-condition:	• The user is authenticated, navigating the budget dashboard, and an expense has already been logged.		
Trigger:	The user clicks the "Edit" action button on a specific expense in the Transaction History table.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user clicks the "Edit" button on a specific expense row.		**Step 2:** The Angular frontend opens the Edit Expense modal, pre-populated with the current amount and category.

	Step 3: The user modifies the details and clicks "Save". [ALT]		**Step 4:** The Angular frontend validates the data and sends a PUT api/Expenses/{id} request with the updated DTO to the backend.
			Step 5: The API controller receives the request and uses FindAsync() to locate the "Expense" record in the database. [ALT]
			Step 6: The system overwrites the relevant properties (AmountZAR, Category) and calls SaveChangesAsync().[ALT]
			Step 7: The API returns an HTTP 204 No Content response.
			Step 8: The Angular frontend closes the modal, refreshes the budget timeline and pie charts, and displays a green success toast.
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel". The modal closes and no changes are made.		
	[ALT] Step 5: The expense cannot be found in the database. The API returns an HTTP 404 Not Found response. The frontend displays a red error toast: "Expense not found."		
	[ALT] Step 6: The database update fails. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast.		
Conclusion:	The expense details are corrected, and the financial graphs update automatically to reflect the accurate amounts.		
Post-condition:	The "Expense" record is modified in the database.		
Business rules:	Users can only edit expenses they logged themselves or must possess administrative permissions.		

Implementation constraints and specifications:	• The "LoggedByName" property is treated as an immutable snapshot and cannot be modified during this update process.
Assumptions:	• The frontend automatically performs basic validation (e.g., preventing negative currency amounts) before sending the request.
Open issues:	• Loading time for the data is slow

Table 135: Update An Expense - Technical Use Case Narrative

2.14.5 View Expense

INTEGRATED NGO MANAGEMENT SYSTEM			
Author:	Alexander Steven	**Date:** 15 July 2026 **Version:** 1.0	
Use case name:	View Expense	**USE CASE TYPE** Business Requirements: □ System Analysis: □ System Design: ✓	
Use case id:	14.5		
Priority:	Medium		
Source:	EPI-USE		
Primary business actor (PBA):	User (Authenticated)		
Primary system actor (PSA):	• System		
Other participating actors:	• None		
Other interested stakeholders:	None		
Description:	A user with appropriate permissions can view the full details of a specific logged expense. When the user clicks on a row in the Transaction History table or clicks "View Receipt", the system opens a detailed modal. Since the expense data was already passed to the frontend in the "ProjectBudgetDetailDto" payload during dashboard load, the Angular client seamlessly renders the details (Date, Category, Amount, Logged By, and the attached receipt image) without making an additional database query.		
Pre-condition:	• The user is authenticated, navigating the budget dashboard, and expenses exist in the Transaction History table.		
Trigger:	The user clicks a specific expense row or the "View Receipt" button in the table.		
Typical Course of Events:	**Actor Action**	**System Response**	
		Manual Action	**Automated Action**
	Step 1: The user clicks "View Receipt" on a specific transaction row.		**Step 2:** The Angular frontend intercepts the click and retrieves the specific "ExpenseItemDto" from its local memory state.

			Step 3: The system launches a detailed view modal and populates it with the expense data, rendering the image from the "ReceiptUrl". [ALT]
	Step 4: The user reviews the details and clicks "Close".		**Step 5:** The Angular frontend closes the modal.
Alternate courses:	**[ALT] Step 3:** The "ReceiptUrl" is empty or the image file cannot be fetched from the server. The frontend displays a placeholder icon and the text "No receipt uploaded."		
Conclusion:	The user successfully views the full details and physical receipt image for a transaction.		
Post-condition:	The modal is closed and the user returns to the main budget dashboard.		
Business rules:	Users can only view expense receipts for projects they have permission to access.		
Implementation constraints and specifications:	• Because the data is already contained in the "ProjectBudgetDetailDto" from Use Case 15.1, this action does not trigger a separate GET request to the .NET API, heavily optimizing server load.		
Assumptions:	• The browser has access to load the image path stored in the "ReceiptUrl".		
Open issues:	• None		

Table 136: View Expense - Technical Use Case Narrative

2.14.6 Remove Expense

<table>
<tr><th colspan="4">INTEGRATED NGO MANAGEMENT SYSTEM</th></tr>
<tr><td>Author:</td><td>Alexander Steven</td><td colspan="2">Date: 15 July 2026
Version: 1.0</td></tr>
<tr><td>Use case name:</td><td>Remove Expense</td><td colspan="2" rowspan="4">USE CASE TYPE
Business Requirements: □
System Analysis: □
System Design: ✓</td></tr>
<tr><td>Use case id:</td><td>14.6</td></tr>
<tr><td>Priority:</td><td>Medium</td></tr>
<tr><td>Source:</td><td>EPI-USE</td></tr>
<tr><td>Primary business actor (PBA):</td><td colspan="3">user (Authenticated)</td></tr>
<tr><td>Primary system actor (PSA):</td><td colspan="3">• None</td></tr>
<tr><td>Other participating actors:</td><td colspan="3">• None</td></tr>
<tr><td>Other interested stakeholders:</td><td colspan="3">None</td></tr>
<tr><td>Description:</td><td colspan="3">A user with appropriate permissions can permanently delete an expense from the project budget. The user clicks the "Delete" button on a transaction row. The Angular frontend displays a confirmation prompt. Upon confirmation, the system sends a DELETE api/Expenses/{id} request to the .NET 8 API. The backend uses Entity Framework Core to locate the "Expense" record, removes the associated receipt file from the server (if applicable), deletes the database record, and saves the changes. The financial charts are then updated to reflect the removed amount.</td></tr>
<tr><td>Pre-condition:</td><td colspan="3">• The user is authenticated, viewing the budget dashboard, and an expense exists.</td></tr>
<tr><td>Trigger:</td><td colspan="3">The user clicks the "Delete" icon/button on an expense row.</td></tr>
<tr><td rowspan="2">Typical Course of Events:</td><td rowspan="2">Actor Action</td><td colspan="2">System Response</td></tr>
<tr><td>Manual Action</td><td>Automated Action</td></tr>
<tr><td>Step 1: The user clicks the "Delete" button on a specific expense.</td><td></td><td>Step 2: The Angular frontend pauses the action and displays a confirmation modal: "Are you sure you want to delete this expense?"</td></tr>
</table>

	Step 3: The user clicks "Confirm". [ALT]		**Step 4:** The Angular frontend sends a DELETE api/Expenses/{id} request to the backend.
			Step 5: The API controller receives the request and locates the "Expense" record in the database using FindAsync(). [ALT]
			Step 6: The system checks if a "ReceiptUrl" exists. If so, it locates the physical file on the server and executes System.IO.File.Delete() to remove it.
			Step 7: The system calls _context.Expenses.Remove() and SaveChangesAsync() to commit the database deletion. [ALT]
			Step 8: The API returns an HTTP 204 No Content response.
			Step 9: The Angular frontend removes the row from the table, recalculates the budget charts dynamically, and displays a green success toast.
Alternate courses:	**[ALT] Step 3:** The user clicks "Cancel". The modal closes and the deletion are aborted.		
	[ALT] Step 5: The expense cannot be found in the database. The API returns an HTTP 404 Not Found response. The frontend displays a red error toast.		
	[ALT] Step 7: The database connection fails during saving. The API returns an HTTP 500 Internal Server Error, and the frontend displays a red error toast: "Failed to delete expense."		
Conclusion:	The erroneous expense is permanently erased, restoring accuracy to the project's financial budget.		

Post-condition:	The "Expense" database record is destroyed, and the physical receipt file (if any) is deleted.
Business rules:	Users can only delete expenses they created themselves, unless they are a System Admin.
Implementation constraints and specifications:	• The physical receipt file must be deleted before the database record is removed to prevent orphaned files consuming storage space.
Assumptions:	• The backend API is programmed with a DELETE endpoint in the ExpensesController that mirrors the standard architecture.
Open issues:	• None

Table 137: Remove Expense - Technical Use Case Narrative

3. Conclusion

This document outlined the technical narratives for the use cases defined for the proposed system for EPI-USE and groupelephant.com.

4. Sign-off by Client

Sign-Off Confirmation:

I, the undersigned, have reviewed the contents of this document and approved it as complete and accurate. I acknowledge that the information provided meets the requirements agreed upon with team 35, Canvas.

Client Name: Dané Coetzee

Signature: Dané Coetzee

Date: 11/08/2026

5. Sign-off by Team

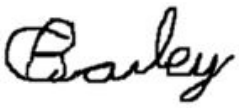
Team member:
Name: Alexander Steven
Date: 16 August 2026
Signature:



Team member:
Name: Amahle Khoza
Date: 16 August 2026
Signature:



Team member:
Name: Christopher Bailey
Date: 16 August 2026
Signature:



Team member:
Name: Nicholas Bailey
Date: 16 August 2026
Signature:



Team member:
Name: Stephan Bredenhann
Date: 16 August 2026
Signature:

