Domain Entities, Enums and Service Interfaces

# 1. Domain Entities & Enums

## AuditActionType.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Audit  
{  
 public enum AuditActionType  
 {  
 Create,  
 Update,  
 Delete  
 }  
  
}

## AuditTrail.cs

using System;  
using App.Core.Domain.Users;  
  
namespace App.Core.Domain.Audit  
{  
 public class AuditTrail : BaseEntity  
 {  
 /// <summary>  
 /// Name of the table/entity that was changed  
 /// </summary>  
 public string EntityName { get; set; }  
  
 /// <summary>  
 /// Primary key of the entity that was changed  
 /// </summary>  
 public int EntityId { get; set; }  
  
 /// <summary>  
 /// Field/column that was changed (if applicable)  
 /// </summary>  
 public string FieldName { get; set; }  
  
 /// <summary>  
 /// Old value of the field  
 /// </summary>  
 public string OldValue { get; set; }  
  
 /// <summary>  
 /// New value of the field  
 /// </summary>  
 public string NewValue { get; set; }  
  
 /// <summary>  
 /// User who made the change  
 /// </summary>  
 public int ChangedByUserId { get; set; }  
 public virtual User ChangedByUser { get; set; }  
  
 /// <summary>  
 /// UTC timestamp when the change occurred  
 /// </summary>  
 public DateTime ChangedOnUtc { get; set; }  
  
 /// <summary>  
 /// Action performed (Create, Update, Delete)  
 /// </summary>  
 public int ActionId { get; set; }  
 public AuditActionType Action { get; set; }  
  
 /// <summary>  
 /// IP address of the client who performed the action  
 /// </summary>  
 public string ClientIp { get; set; }  
  
 /// <summary>  
 /// Optional comment/notes about the change  
 /// </summary>  
 public string Comment { get; set; }  
 }  
}

## AuditTrailField.cs

namespace App.Core.Domain.Audit  
{  
 /// <summary>  
 /// Lookup for audited fields within a table.  
 /// </summary>  
 public class AuditTrailField : BaseEntity  
 {  
 public int AuditTrailTableId { get; set; }  
 public string DBFieldName { get; set; }  
 public string SystemName { get; set; }  
 }  
}

## AuditTrailTable.cs

namespace App.Core.Domain.Audit  
{  
 /// <summary>  
 /// Lookup for audited database tables.  
 /// </summary>  
 public class AuditTrailTable : BaseEntity  
 {  
 public string DBName { get; set; }  
 public string SystemName { get; set; } // used as key for localization  
 }  
}

## Attachment.cs

using System;  
  
namespace App.Core.Domain.Common  
{  
 /// <summary>  
 /// Represents a file attachment (used for documents, IDs, etc.).  
 /// </summary>  
 public class Attachment : BaseEntity  
 {  
 public string FileName { get; set; }  
 public string StoragePath { get; set; }  
 public string MimeType { get; set; }  
 public string Extension { get; set; }  
 public long FileSize { get; set; }  
  
 public int UploadedByUserId { get; set; }  
 public DateTime UploadedOnUtc { get; set; }  
 }  
}

## CompositeKeyAttribute.cs

﻿using System;  
  
namespace App.Core.Domain.Common  
{  
 /// <summary>  
 /// Define composite primary key for a mapping table  
 /// </summary>  
 [AttributeUsage(AttributeTargets.Class)]  
 public partial class CompositeKeyAttribute : Attribute  
 {  
 public string[] PropertyNames { get; }  
  
 public CompositeKeyAttribute(params string[] propertyNames)  
 {  
 PropertyNames = propertyNames;  
 }  
 }  
}

## GenericAttribute.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Common  
{  
 /// <summary>  
 /// Represents a generic attribute  
 /// </summary>  
 public partial class GenericAttribute : BaseEntity  
 {  
 /// <summary>  
 /// Gets or sets the entity identifier  
 /// </summary>  
 public int EntityId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the key group  
 /// </summary>  
 public string KeyGroup { get; set; }  
  
 /// <summary>  
 /// Gets or sets the key  
 /// </summary>  
 public string Key { get; set; }  
  
 /// <summary>  
 /// Gets or sets the value  
 /// </summary>  
 public string Value { get; set; }  
  
 /// <summary>  
 /// Gets or sets the store identifier  
 /// </summary>  
 public int StoreId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the created or updated date  
 /// </summary>  
 public DateTime? CreatedOrUpdatedDateUTC { get; set; }  
 }  
}

## SkipIdAttribute.cs

﻿using System;  
  
namespace App.Core.Domain.Common  
{  
 /// <summary>  
 /// Use this to tell the auto-schema generator to skip creating the default Id column  
 /// </summary>  
 [AttributeUsage(AttributeTargets.Class)]  
 public partial class SkipIdAttribute : Attribute  
 {  
 }  
}

## Template.cs

using System;  
using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Common  
{  
 public partial class Template : BaseEntity  
 {  
 public string Name { get; set; }  
 public string TemplateType { get; set; } // Email, SMS, Document  
 public string Content { get; set; }  
 public int CreatedByUserId { get; set; }  
 public DateTime CreatedOnUtc { get; set; }  
 }  
}

## Correspondence.cs

using System;  
using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Correspondences  
{  
 public partial class Correspondence : BaseEntity  
 {  
 public int SenderUserId { get; set; }  
 public int RecipientUserId { get; set; }  
 public string Subject { get; set; }  
 public string Message { get; set; }  
 public DateTime SentOnUtc { get; set; }  
 public string Status { get; set; } // Sent, Delivered, Read, Archived  
 }  
}

## CalendarEvent.cs

using System;  
using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Directory  
{  
 public partial class CalendarEvent : BaseEntity  
 {  
 public string Title { get; set; }  
 public string Description { get; set; }  
 public DateTime StartDateUtc { get; set; }  
 public DateTime EndDateUtc { get; set; }  
 public bool IsHoliday { get; set; }  
 public string EventType { get; set; } // PublicHoliday, Deadline, Meeting  
 public int CreatedByUserId { get; set; }  
 }  
}

## Country.cs

﻿using App.Core;  
  
namespace App.Core.Domain.Directory;  
  
/// <summary>  
/// Represents a country  
/// </summary>  
public partial class Country : BaseEntity  
{  
 /// <summary>  
 /// Gets or sets the name  
 /// </summary>  
 public string Name { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether billing is allowed to this country  
 /// </summary>  
 public bool AllowsBilling { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether shipping is allowed to this country  
 /// </summary>  
 public bool AllowsShipping { get; set; }  
  
 /// <summary>  
 /// Gets or sets the two letter ISO code  
 /// </summary>  
 public string TwoLetterIsoCode { get; set; }  
  
 /// <summary>  
 /// Gets or sets the three letter ISO code  
 /// </summary>  
 public string ThreeLetterIsoCode { get; set; }  
  
 /// <summary>  
 /// Gets or sets the numeric ISO code  
 /// </summary>  
 public int NumericIsoCode { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether customers in this country must be charged EU VAT  
 /// </summary>  
 public bool SubjectToVat { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether the entity is published  
 /// </summary>  
 public bool Published { get; set; }  
  
 /// <summary>  
 /// Gets or sets the display order  
 /// </summary>  
 public int DisplayOrder { get; set; }  
}

## StateProvince.cs

﻿using App.Core;  
  
namespace App.Core.Domain.Directory;  
  
/// <summary>  
/// Represents a state/province  
/// </summary>  
public partial class StateProvince : BaseEntity  
{  
 /// <summary>  
 /// Gets or sets the country identifier  
 /// </summary>  
 public int CountryId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the name  
 /// </summary>  
 public string Name { get; set; }  
  
 /// <summary>  
 /// Gets or sets the abbreviation  
 /// </summary>  
 public string Abbreviation { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether the entity is published  
 /// </summary>  
 public bool Published { get; set; }  
  
 /// <summary>  
 /// Gets or sets the display order  
 /// </summary>  
 public int DisplayOrder { get; set; }  
}

## WorkingLocation.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Directory  
{  
 public partial class WorkingLocation : BaseEntity  
 {  
 public string Name { get; set; }  
 public string Address { get; set; }  
 public int CountryId { get; set; }  
 }  
}

## Contract.cs

namespace App.Core.Domain.Institutions  
{  
 /// <summary>  
 /// Contract type entity (per Appendix).  
 /// </summary>  
 public class Contract : BaseEntity  
 {  
 public string Name { get; set; }  
 public string Code { get; set; }  
 public bool IsActive { get; set; }  
 }  
}

## Department.cs

namespace App.Core.Domain.Institutions  
{  
 public class Department : BaseEntity  
 {  
 public string Name { get; set; }  
 public int? ParentId { get; set; }  
 public bool IsActive { get; set; }  
  
 // New field for manager  
 public int? ManagerId { get; set; }  
 }  
}

## Duty.cs

using System;  
  
namespace App.Core.Domain.Institutions  
{  
 /// <summary>  
 /// Official duty assigned to an employee (per Appendix).  
 /// </summary>  
 public class Duty : BaseEntity  
 {  
 public int EmployeeId { get; set; }  
 public string DutyType { get; set; }  
 public DateTime FromDate { get; set; }  
 public DateTime ToDate { get; set; }  
 public string Notes { get; set; }  
 }  
}

## Institution.cs

using System;  
using System.Collections.Generic;  
using App.Core.Domain.Ref;  
  
namespace App.Core.Domain.Institutions  
{  
 /// <summary>  
 /// Represents a financial institution (final approved entity after registration).  
 /// </summary>  
 public class Institution : BaseEntity  
 {  
 /// <summary>  
 /// Institution official name  
 /// </summary>  
 public string Name { get; set; }  
  
 /// <summary>  
 /// License number assigned to the institution  
 /// </summary>  
 public string LicenseNumber { get; set; }  
  
 /// <summary>  
 /// License type (Islamic, Commercial, FinTech)  
 /// </summary>  
 public int LicenseTypeId { get; set; }  
 public LicenseType LicenseType { get; set; }  
  
 /// <summary>  
 /// License sector (Banking / Exchange / Insurance)  
 /// </summary>  
 public int LicenseSectorId { get; set; }  
 public LicenseSector LicenseSector { get; set; }  
  
 /// <summary>  
 /// Financial domain (Islamic / Commercial)  
 /// </summary>  
 public int FinancialDomainId { get; set; }  
 public FinancialDomain FinancialDomain { get; set; }  
  
 /// <summary>  
 /// License issue date  
 /// </summary>  
 public DateTime LicenseIssueDate { get; set; }  
  
 /// <summary>  
 /// License expiry date  
 /// </summary>  
 public DateTime LicenseExpiryDate { get; set; }  
  
 /// <summary>  
 /// Country reference  
 /// </summary>  
 public int CountryId { get; set; }  
  
 /// <summary>  
 /// Address of the institution  
 /// </summary>  
 public string Address { get; set; }  
  
 /// <summary>  
 /// Is the institution active  
 /// </summary>  
 public bool IsActive { get; set; }  
  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? UpdatedOnUtc { get; set; }  
  
 // Navigation  
 public virtual ICollection<Department> Departments { get; set; }  
 public virtual ICollection<Job> Jobs { get; set; }  
 public virtual ICollection<Contract> Contracts { get; set; }  
 public virtual ICollection<Duty> Duties { get; set; }  
 }  
}

## Job.cs

namespace App.Core.Domain.Institutions  
{  
 /// <summary>  
 /// Job title entity (per Appendix).  
 /// </summary>  
 public class Job : BaseEntity  
 {  
 public string Name { get; set; }  
 public bool IsActive { get; set; }  
 }  
}

## Language.cs

﻿using System;  
using System.ComponentModel.DataAnnotations;  
using App.Core;  
  
namespace App.Core.Domain.Localization  
{  
 /// <summary>  
 /// Represents a language  
 /// </summary>  
 public partial class Language : BaseEntity  
 {  
 /// <summary>  
 /// Gets or sets the name  
 /// </summary>  
 public string Name { get; set; }  
  
 /// <summary>  
 /// Gets or sets the language culture  
 /// </summary>  
 public string LanguageCulture { get; set; }  
  
 /// <summary>  
 /// Gets or sets the unique SEO code  
 /// </summary>  
 public string UniqueSeoCode { get; set; }  
  
 /// <summary>  
 /// Gets or sets the flag image file name  
 /// </summary>  
 public string FlagImageFileName { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether the language supports "Right-to-left"  
 /// </summary>  
 public bool Rtl { get; set; }  
  
 /// <summary>  
 /// Gets or sets a value indicating whether the language is published  
 /// </summary>  
 public bool Published { get; set; }  
  
 /// <summary>  
 /// Gets or sets the display order  
 /// </summary>  
 public int DisplayOrder { get; set; }  
 }  
}

## LocaleStringResource.cs

﻿using App.Core;  
using System;  
using System.Collections.Generic;  
using System.ComponentModel.DataAnnotations;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Localization  
{  
 /// <summary>  
 /// Represents a locale string resource  
 /// </summary>  
 public partial class LocaleStringResource : BaseEntity  
 {  
 /// <summary>  
 /// Gets or sets the language identifier  
 /// </summary>  
 public int LanguageId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the resource name  
 /// </summary>  
 public string ResourceName { get; set; }  
  
 /// <summary>  
 /// Gets or sets the resource value  
 /// </summary>  
 public string? ResourceValue { get; set; }  
 }  
}

## LocalizationSettings.cs

﻿using App.Core.Configuration;  
  
namespace App.Core.Domain.Localization;  
  
/// <summary>  
/// Localization settings  
/// </summary>  
public partial class LocalizationSettings : BaseEntity, ISettings  
{  
 /// <summary>  
 /// Default admin area language identifier  
 /// </summary>  
 public int DefaultAdminLanguageId { get; set; }  
  
 /// <summary>  
 /// Use images for language selection  
 /// </summary>  
 public bool UseImagesForLanguageSelection { get; set; }  
  
 /// <summary>  
 /// A value indicating whether SEO friendly URLs with multiple languages are enabled  
 /// </summary>  
 public bool SeoFriendlyUrlsForLanguagesEnabled { get; set; }  
  
 /// <summary>  
 /// A value indicating whether we should detect the current language by a customer region (browser settings)  
 /// </summary>  
 public bool AutomaticallyDetectLanguage { get; set; }  
  
 /// <summary>  
 /// A value indicating whether to load all LocaleStringResource records on application startup  
 /// </summary>  
 public bool LoadAllLocaleRecordsOnStartup { get; set; }  
  
 /// <summary>  
 /// A value indicating whether to load all LocalizedProperty records on application startup  
 /// </summary>  
 public bool LoadAllLocalizedPropertiesOnStartup { get; set; }  
  
 /// <summary>  
 /// A value indicating whether to load all search engine friendly names (slugs) on application startup  
 /// </summary>  
 public bool LoadAllUrlRecordsOnStartup { get; set; }  
  
 /// <summary>  
 /// A value indicating whether to we should ignore RTL language property for admin area.  
 /// It's useful for store owners with RTL languages for public store but preferring LTR for admin area  
 /// </summary>  
 public bool IgnoreRtlPropertyForAdminArea { get; set; }  
}

## LocalizedProperty.cs

﻿using System.ComponentModel.DataAnnotations;  
using App.Core;  
  
namespace App.Core.Domain.Localization  
{  
 /// <summary>  
 /// Represents a localized property  
 /// </summary>  
 public partial class LocalizedProperty : BaseEntity  
 {  
 /// <summary>  
 /// Gets or sets the entity identifier  
 /// </summary>  
 public int EntityId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the language identifier  
 /// </summary>  
 public int LanguageId { get; set; }  
  
 /// <summary>  
 /// Gets or sets the locale key group  
 /// </summary>  
 public string LocaleKeyGroup { get; set; }  
  
 /// <summary>  
 /// Gets or sets the locale key  
 /// </summary>  
 public string LocaleKey { get; set; }  
  
 /// <summary>  
 /// Gets or sets the locale value  
 /// </summary>  
 public string LocaleValue { get; set; }  
 }  
}

## Notification.cs

using System;  
using App.Core.Domain.Common;  
using App.Core.Domain.Registrations;  
using App.Core.Domain.Users;  
  
namespace App.Core.Domain.Notifications  
{  
 public partial class Notification : BaseEntity  
 {  
 /// <summary>  
 /// Related registration (if applicable)  
 /// </summary>  
 public int? RegistrationId { get; set; }  
 public virtual Registration Registration { get; set; }  
  
 /// <summary>  
 /// The type of event that triggered this notification  
 /// </summary>  
 public int EventTypeId { get; set; }  
 public NotificationEvent EventType { get; set; }  
  
 /// <summary>  
 /// The recipient user  
 /// </summary>  
 public int RecipientUserId { get; set; }  
 public virtual User RecipientUser { get; set; }  
  
 /// <summary>  
 /// The user who triggered the notification  
 /// </summary>  
 public int TriggeredByUserId { get; set; }  
 public virtual User TriggeredByUser { get; set; }  
  
 /// <summary>  
 /// The notification message body  
 /// </summary>  
 public string Message { get; set; }  
  
 /// <summary>  
 /// Delivery channel (Email, SMS, InApp, Push)  
 /// </summary>  
 public int ChannelId { get; set; }  
 public NotificationChannel Channel { get; set; }  
  
 /// <summary>  
 /// Delivery status (Pending, Sent, Failed)  
 /// </summary>  
 public int StatusId { get; set; }  
 public NotificationDeliveryStatus Status { get; set; }  
  
 /// <summary>  
 /// UTC timestamp when the notification was created  
 /// </summary>  
 public DateTime CreatedOnUtc { get; set; }  
 }  
}

## NotificationChannel.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Notifications  
{  
 public enum NotificationChannel  
 {  
 InApp = 0,  
 Email = 1,  
 SMS = 2,  
 Push = 3  
 }  
}

## NotificationDeliveryStatus.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Notifications  
{  
 public enum NotificationDeliveryStatus  
 {  
 Pending = 0,  
 Sent = 1,  
 Failed = 2  
 }  
}

## NotificationEvent.cs

﻿namespace App.Core.Domain.Notifications  
{  
 /// <summary>  
 /// Defines system notification events as per BRD.  
 /// </summary>  
 public enum NotificationEvent  
 {  
 // Workflow - Draft/Submission  
 RegistrationSubmitted = 1,  
  
 // Validation  
 RegistrationValidated = 2,  
  
 // Approval  
 RegistrationApproved = 3,  
 RegistrationRejected = 4,  
 RegistrationReturnedForEdit = 5,  
  
 // Audit  
 RegistrationAudited = 6,  
  
 // Final  
 RegistrationFinalSubmission = 7,  
 RegistrationArchived = 8,  
  
 // Assignment  
 NewAssignment = 9  
 }  
}

## NotificationLog.cs

using System;  
  
namespace App.Core.Domain.Notifications  
{  
 /// <summary>  
 /// Detailed notification delivery log (email, sms, etc.)  
 /// </summary>  
 public class NotificationLog : BaseEntity  
 {  
 public int NotificationId { get; set; }  
 public string Channel { get; set; }  
 public bool Success { get; set; }  
 public string Response { get; set; }  
 public DateTime SentOnUtc { get; set; }  
 }  
}

## NotificationReadLog.cs

using System;  
  
namespace App.Core.Domain.Notifications  
{  
 /// <summary>  
 /// Tracks when a user reads/dismisses a notification.  
 /// </summary>  
 public class NotificationReadLog : BaseEntity  
 {  
 public int NotificationId { get; set; }  
 public int UserId { get; set; }  
 public DateTime ReadOnUtc { get; set; }  
 }  
}

## BusinessScaleRange.cs

namespace App.Core.Domain.Ref  
{  
 /// <summary>  
 /// Lookup for institution business scale ranges (per BRD Appendix).  
 /// </summary>  
 public class BusinessScaleRange : BaseEntity  
 {  
 public string Name { get; set; }  
 public int MinValue { get; set; }  
 public int MaxValue { get; set; }  
 public string RangeLabel { get; set; }  
 }  
}

## EmployeeRange.cs

namespace App.Core.Domain.Ref  
{  
 /// <summary>  
 /// Lookup for institution employee count ranges (per BRD Appendix).  
 /// </summary>  
 public class EmployeeRange : BaseEntity  
 {  
 public string Name { get; set; }  
 public int MinValue { get; set; }  
 public int MaxValue { get; set; }  
 }  
}

## FinancialDomain.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Ref  
{  
 /// <summary>  
 /// Defines financial domain types per BRD.  
 /// </summary>  
 public enum FinancialDomain  
 {  
 Islamic = 1,  
 Commercial = 2  
 }  
}

## LicenseSector.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Ref  
{  
 /// <summary>  
 /// Defines license sectors per BRD.  
 /// </summary>  
 public enum LicenseSector  
 {  
 /// <summary>  
 /// Banking sector  
 /// </summary>  
 Banking = 1,  
  
 /// <summary>  
 /// Exchange sector  
 /// </summary>  
 Exchange = 2  
 }  
}

## LicenseType.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Ref  
{  
 public enum LicenseType  
 {  
 /// <summary>  
 /// Islamic financial license  
 /// </summary>  
 Islamic = 1,  
  
 /// <summary>  
 /// Commercial financial license  
 /// </summary>  
 Commercial = 2,  
  
 /// <summary>  
 /// Banking license  
 /// </summary>  
 Banking = 3,  
  
 /// <summary>  
 /// Exchange license  
 /// </summary>  
 Exchange = 4,  
  
 /// <summary>  
 /// FinTech / Technology-based financial license  
 /// </summary>  
 FinTech = 5  
 }  
}

## ApprovalStatus.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Approval outcomes as per BRD.  
 /// </summary>  
 public enum ApprovalStatus  
 {  
 Accepted = 0,  
 Return = 1,  
 Rejected = 2  
 }  
}

## AuditStatus.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Audit outcomes as per BRD.  
 /// </summary>  
 public enum AuditStatus  
 {  
 Accepted = 0,  
 Return = 1,  
 Rejected = 2  
 }  
}

## ContactType.cs

namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Contact types per BRD.  
 /// </summary>  
 public enum ContactType  
 {  
 Primary = 0,  
 Authorized = 1,  
 Delegate = 2,  
 BusinessOwner = 3  
 }  
}

## FIContact.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Registrations  
{  
 public partial class FIContact : BaseEntity  
 {  
 public int RegistrationId { get; set; }  
  
 /// <summary>  
 /// Type of contact (Primary, Authorized, Delegate, BusinessOwner)  
 /// </summary>  
 public int ContactTypeId { get; set; }  
 public ContactType ContactType { get; set; }  
  
 public string JobTitle { get; set; }  
  
 public string FirstName { get; set; }  
 public string MiddleName { get; set; }  
 public string LastName { get; set; }  
  
 public string CivilId { get; set; }  
 public string PassportId { get; set; }  
  
 public int CivilAttachmentId { get; set; }  
 public int PassportAttachmentId { get; set; }  
  
 public string ContactPhone { get; set; }  
 public string BusinessPhone { get; set; }  
 public string Email { get; set; }  
  
 public int NationalityCountryId { get; set; }  
  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? UpdatedOnUtc { get; set; }  
 }  
}

## FIDocument.cs

using System;  
using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Registrations  
{  
 public partial class FIDocument : BaseEntity  
 {  
 public int RegistrationId { get; set; }  
 public string DocumentType { get; set; }  
 public string FilePath { get; set; }  
 public int UploadedByUserId { get; set; }  
 public DateTime UploadedOnUtc { get; set; }  
 }  
}

## FIRegistrationStatusLog.cs

using System;  
  
namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Tracks workflow transitions & status history for a registration.  
 /// </summary>  
 public class FIRegistrationStatusLog : BaseEntity  
 {  
 /// <summary>  
 /// The main registration record  
 /// </summary>  
 public int RegistrationId { get; set; }  
  
 /// <summary>  
 /// The overall lifecycle status  
 /// </summary>  
 public RegistrationStatus RegistrationStatus { get; set; }  
  
 /// <summary>  
 /// Optional validation step result  
 /// </summary>  
 public ValidationStatus? ValidationStatus { get; set; }  
  
 /// <summary>  
 /// Optional approval step result  
 /// </summary>  
 public ApprovalStatus? ApprovalStatus { get; set; }  
  
 /// <summary>  
 /// Optional audit step result  
 /// </summary>  
 public AuditStatus? AuditStatus { get; set; }  
  
 /// <summary>  
 /// Who performed the action (UserId)  
 /// </summary>  
 public int PerformedBy { get; set; }  
  
 /// <summary>  
 /// Timestamp of the action  
 /// </summary>  
 public DateTime ActionDateUtc { get; set; }  
  
 /// <summary>  
 /// Optional remarks or reason for status change  
 /// </summary>  
 public string Remarks { get; set; }  
 }  
}

## Registration.cs

using System;  
using App.Core.Domain.Ref;  
using App.Core.Domain.Institutions;  
  
namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Represents a registration record (snapshot + lifecycle).  
 /// </summary>  
 public partial class Registration : BaseEntity  
 {  
 /// <summary>  
 /// Institution reference  
 /// </summary>  
 public int InstitutionId { get; set; }  
 public virtual Institution Institution { get; set; }  
  
 /// <summary>  
 /// Institution name snapshot (copied from Institution at submission time)  
 /// </summary>  
 public string InstitutionName { get; set; }  
  
 /// <summary>  
 /// License number snapshot  
 /// </summary>  
 public string LicenseNumber { get; set; }  
  
 /// <summary>  
 /// License sector (Banking / Exchange)  
 /// </summary>  
 public int LicenseSectorId { get; set; }  
 public LicenseSector LicenseSector { get; set; }  
  
 /// <summary>  
 /// License type (Islamic, Commercial, FinTech)  
 /// </summary>  
 public int LicenseTypeId { get; set; }  
 public LicenseType LicenseType { get; set; }  
  
 /// <summary>  
 /// Financial domain (Islamic / Commercial)  
 /// </summary>  
 public int FinancialDomainId { get; set; }  
 public FinancialDomain FinancialDomain { get; set; }  
  
 /// <summary>  
 /// License issue date  
 /// </summary>  
 public DateTime IssueDate { get; set; }  
  
 /// <summary>  
 /// License expiry date  
 /// </summary>  
 public DateTime ExpiryDate { get; set; }  
  
 /// <summary>  
 /// Country of the institution  
 /// </summary>  
 public int CountryId { get; set; }  
  
 /// <summary>  
 /// Address of the institution  
 /// </summary>  
 public string Address { get; set; }  
  
 /// <summary>  
 /// Main lifecycle status of registration  
 /// </summary>  
 ///   
 public int StatusId { get; set; }  
 public RegistrationStatus Status { get; set; }  
  
 public int CreatedByUserId { get; set; }  
 public int? UpdatedByUserId { get; set; }  
 public int? SubmittedToUserId { get; set; }  
  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? SubmittedDateUtc { get; set; }  
 public DateTime? ApprovedDateUtc { get; set; }  
 public DateTime? AuditedDateUtc { get; set; }  
  
 public int? BusinessScaleRangeId { get; set; }  
 public int? EmployeeRangeId { get; set; }  
 }  
}

## RegistrationAction.cs

﻿namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Defines available actions for a registration.  
 /// </summary>  
 public enum RegistrationAction  
 {  
 Submit,  
 Validate,  
 Approve,  
 Reject,  
 ReturnForEdit,  
 Audit,  
 Archive,  
 FinalSubmission  
 }  
}

## RegistrationStatus.cs

namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Registration lifecycle per BRD.  
 /// </summary>  
 public enum RegistrationStatus  
 {  
 Draft = 0,  
 Submitted = 1,  
 UnderReview = 2,  
 Approved = 3,  
 Rejected = 4,  
 ReturnedForEdit = 5,  
 Archived = 6,  
 FinalSubmission = 7  
 }  
}

## ValidationStatus.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Registrations  
{  
 /// <summary>  
 /// Validation lifecycle as per BRD.  
 /// </summary>  
 public enum ValidationStatus  
 {  
 Draft = 0,  
 SubmitForApproval = 1,  
 Accepted = 2,  
 Return = 3,  
 Rejected = 4  
 }  
}

## WorkflowStep.cs

using App.Core.Domain.Common;  
using App.Core.Domain.Security;  
  
namespace App.Core.Domain.Registrations  
{  
 public partial class WorkflowStep : BaseEntity  
 {  
 public string Name { get; set; }  
 public int? NextStepId { get; set; }  
  
 public int RoleAllowedId { get; set; }  
 public Role RoleAllowed { get; set; }  
 }  
}

## Dashboard.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Reports  
{  
 public partial class Dashboard : BaseEntity  
 {  
 public string Name { get; set; }  
 public string Description { get; set; }  
 }  
}

## Report.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Reports  
{  
 public partial class Report : BaseEntity  
 {  
 public string Title { get; set; }  
 public string ReportType { get; set; }  
 public string FilePath { get; set; }  
 public DateTime CreatedOnUtc { get; set; }  
 }  
}

## AppPermissions.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Security  
{  
 /// <summary>  
 /// Defines all system permission constants.  
 /// These values are used as unique keys in the ACL system.  
 /// </summary>  
 public static class AppPermissions  
 {  
 // Registration  
 public const string Registration\_Create = "Registration.Create";  
 public const string Registration\_Read = "Registration.Read";  
 public const string Registration\_Update = "Registration.Update";  
 public const string Registration\_Delete = "Registration.Delete";  
 public const string Registration\_Submit = "Registration.Submit";  
 public const string Registration\_Validate = "Registration.Validate";  
 public const string Registration\_Approve = "Registration.Approve";  
 public const string Registration\_Audit = "Registration.Audit";  
 public const string Registration\_Archive = "Registration.Archive";  
 public const string Registration\_ReturnForEdit = "Registration.ReturnForEdit";  
 public const string Registration\_Reject = "Registration.Reject";  
  
 // Institution  
 public const string Institution\_Create = "Institution.Create";  
 public const string Institution\_Read = "Institution.Read";  
 public const string Institution\_Update = "Institution.Update";  
 public const string Institution\_Delete = "Institution.Delete";  
  
 // Notifications  
 public const string Notification\_Send = "Notification.Send";  
  
 // Directory (Countries, States)  
 public const string Directory\_Country\_Read = "Directory.Country.Read";  
 public const string Directory\_Country\_Update = "Directory.Country.Update";  
 public const string Directory\_State\_Read = "Directory.State.Read";  
 public const string Directory\_State\_Update = "Directory.State.Update";  
  
 // Users  
 public const string User\_Create = "User.Create";  
 public const string User\_Read = "User.Read";  
 public const string User\_Update = "User.Update";  
 public const string User\_Delete = "User.Delete";  
  
 // Roles & Permissions  
 public const string Role\_Manage = "Role.Manage";  
 public const string Permission\_Manage = "Permission.Manage";  
  
 // Admin  
 public const string System\_Admin = "System.Admin";  
 }  
}

## Permission.cs

using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Security  
{  
 public partial class Permission : BaseEntity  
 {  
 public string SystemName { get; set; }  
 public string Category { get; set; }  
 public string Description { get; set; }  
 public bool IsActive { get; set; }  
 }  
}

## Privilege.cs

namespace App.Core.Domain.Security  
{  
 /// <summary>  
 /// Fine-grained permission (controller/action mapping for RBAC).  
 /// </summary>  
 public class Privilege : BaseEntity  
 {  
 public string SystemName { get; set; }  
 public string DisplayName { get; set; }  
 public string Description { get; set; }  
 }  
}

## PrivilegePermission.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Security  
{  
 public class PrivilegePermission : BaseEntity  
 {  
 public int PrivilegeId { get; set; }  
 public int PermissionId { get; set; }  
 }  
}

## Role.cs

using System;  
  
namespace App.Core.Domain.Security  
{  
 /// <summary>  
 /// System role (e.g., Maker, Checker, Regulator, Auditor, Admin).  
 /// </summary>  
 public class Role : BaseEntity  
 {  
 public string Name { get; set; }  
 public string Description { get; set; }  
 public bool IsActive { get; set; }  
  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? UpdatedOnUtc { get; set; }  
 public string SystemName { get; set; }  
 }  
}

## RolePermission .cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Security  
{  
 public class RolePermission : BaseEntity  
 {  
 public int RoleId { get; set; }  
 public int PermissionId { get; set; }  
 }  
}

## RolePrivilege.cs

using System;  
  
namespace App.Core.Domain.Security  
{  
 /// <summary>  
 /// Many-to-many mapping between roles and privileges.  
 /// </summary>  
 public class RolePrivilege : BaseEntity  
 {  
 public int RoleId { get; set; }  
 public int PrivilegeId { get; set; }  
 }  
}

## UserPermissionOverride.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Security  
{  
 public class UserPermissionOverride : BaseEntity  
 {  
 public int UserId { get; set; }  
 public int PermissionId { get; set; }  
 public bool IsGranted { get; set; }  
 }  
}

## Setting.cs

using App.Core;  
using App.Core.Localization;  
using App.Core.Domain;  
  
namespace App.Core.Domain.Settings  
{  
 /// <summary>  
 /// Represents a setting  
 /// </summary>  
 public partial class Setting : BaseEntity, ILocalizedEntity  
 {  
 #region Ctor  
  
 /// <summary>  
 /// Default constructor  
 /// </summary>  
 public Setting()  
 {  
 }  
  
 /// <summary>  
 /// Parameterized constructor  
 /// </summary>  
 /// <param name="name">Setting name</param>  
 /// <param name="value">Setting value</param>  
 /// <param name="storeId">Store identifier</param>  
 public Setting(string name, string value, int storeId = 0)  
 {  
 Name = name;  
 Value = value;  
 StoreId = storeId;  
 }  
  
 #endregion  
  
 #region Properties  
  
 /// <summary>  
 /// Gets or sets the setting name  
 /// </summary>  
 public string Name { get; set; }  
  
 /// <summary>  
 /// Gets or sets the setting value  
 /// </summary>  
 public string Value { get; set; }  
  
 /// <summary>  
 /// Gets or sets the store identifier for which this setting is valid; 0 means all stores  
 /// </summary>  
 public int StoreId { get; set; }  
  
 #endregion  
 }  
}

## PasswordFormat.cs

namespace App.Core.Domain.Users  
{  
 /// <summary>  
 /// Storage format of user passwords.  
 /// </summary>  
 public enum PasswordFormat  
 {  
 Clear = 0,  
 Hashed = 1,  
 Encrypted = 2  
 }  
}

## User.cs

using System;  
using App.Core.Domain.Common;  
  
namespace App.Core.Domain.Users  
{  
 /// <summary>  
 /// Represents a system user (internal or external).  
 /// </summary>  
 public partial class User : BaseEntity  
 {  
 public string Username { get; set; }  
 public string Email { get; set; }  
  
 // Passwords  
 public string PasswordHash { get; set; }  
 public string PasswordSalt { get; set; } // legacy / PBKDF2 salt  
 public int PasswordFormatId { get; set; } = (int)PasswordFormat.Hashed;  
 public PasswordFormat PasswordFormat  
 {  
 get => (PasswordFormat)PasswordFormatId;  
 set => PasswordFormatId = (int)value;  
 }  
  
 // Security  
 public bool IsActive { get; set; }  
 public bool IsLockedOut { get; set; }  
 public int FailedLoginAttempts { get; set; }  
 public DateTime? LockoutEndUtc { get; set; }  
  
 // Audit  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? UpdatedOnUtc { get; set; }  
 public DateTime? LastLoginDateUtc { get; set; }  
 public DateTime? DeactivatedOnUtc { get; set; }  
  
 // Relations  
 public int? InstitutionId { get; set; }  
 public int? RegistrationId { get; set; }  
  
 // Recovery / MFA  
 public string PasswordResetToken { get; set; }  
 public DateTime? PasswordResetTokenExpiry { get; set; }  
  
 public string TwoFactorSecret { get; set; } // e.g., TOTP key  
 public bool TwoFactorEnabled { get; set; }  
 }  
}

## UserLog.cs

using System;  
  
namespace App.Core.Domain.Users  
{  
 /// <summary>  
 /// Logs user-related actions (activation, deactivation, login, password reset).  
 /// </summary>  
 public class UserLog : BaseEntity  
 {  
 public int UserId { get; set; }  
 public string Action { get; set; } // e.g., "Activated", "Deactivated", "PasswordReset"  
 public DateTime CreatedOnUtc { get; set; }  
 public string ClientIp { get; set; }  
 }  
}

## UserPreference.cs

﻿using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Core.Domain.Users  
{  
 namespace App.Core.Domain.Users  
 {  
 public class UserPreference : BaseEntity  
 {  
 public int UserId { get; set; }  
  
 // UI & Localization  
 public string Language { get; set; }  
 public int? LanguageId { get; set; }  
  
 // Security  
 public bool EnableMfa { get; set; }  
  
 // Notifications  
 public bool NotifyByEmail { get; set; }  
 public bool NotifyBySms { get; set; }  
 public bool NotifyInApp { get; set; }  
  
 public DateTime UpdatedOnUtc { get; set; }  
 }  
 }  
  
}

## UserRole.cs

using System;  
  
namespace App.Core.Domain.Users  
{  
 /// <summary>  
 /// Many-to-many mapping between users and roles.  
 /// </summary>  
 public class UserRole : BaseEntity  
 {  
 public int UserId { get; set; }  
 public int RoleId { get; set; }  
  
 public DateTime CreatedOnUtc { get; set; }  
 public DateTime? UpdatedOnUtc { get; set; }  
 }  
}

## UserSettings.cs

namespace App.Core.Domain.Users  
{  
 /// <summary>  
 /// Strongly-typed settings related to user management  
 /// </summary>  
 public class UserSettings  
 {  
 // Password policy  
 public int PasswordMinLength { get; set; } = 6;  
 public bool RequireDigit { get; set; } = true;  
 public bool RequireUppercase { get; set; } = false;  
 public bool RequireLowercase { get; set; } = false;  
 public bool RequireNonAlphanumeric { get; set; } = false;  
  
 // Registration / login  
 public bool RequireUniqueEmail { get; set; } = true;  
 public bool RequireConfirmedEmail { get; set; } = false;  
 public int MaxFailedAccessAttempts { get; set; } = 5;  
 public int DefaultLockoutMinutes { get; set; } = 15;  
  
 // Profile  
 public bool AllowProfileEditing { get; set; } = true;  
 public string DefaultAvatarUrl { get; set; } = null;  
 }  
}

# 2. Services & Interfaces

## Extensions.cs

﻿using Microsoft.AspNetCore.Mvc.Rendering;  
using App.Core;  
using App.Core.Infrastructure;  
using App.Core.Domain;  
using App.Core.EngineServices;  
using App.Services.Localization;  
using App.Data.EngineServices;  
  
namespace App.Services;  
  
/// <summary>  
/// Extensions  
/// </summary>  
public static class Extensions  
{  
 /// <summary>  
 /// Convert to select list  
 /// </summary>  
 /// <typeparam name="TEnum">Enum type</typeparam>  
 /// <param name="enumObj">Enum</param>  
 /// <param name="markCurrentAsSelected">Mark current value as selected</param>  
 /// <param name="valuesToExclude">Values to exclude</param>  
 /// <param name="useLocalization">Localize</param>  
 /// <returns>  
 /// A task that represents the asynchronous operation  
 /// The task result contains the selectList  
 /// </returns>  
 public static async Task<SelectList> ToSelectListAsync<TEnum>(this TEnum enumObj,  
 bool markCurrentAsSelected = true, int[] valuesToExclude = null, bool useLocalization = true) where TEnum : Enum  
 {  
 if (enumObj == null)  
 throw new ArgumentNullException(nameof(enumObj));  
  
 var localizationService = EngineContext.Current.Resolve<ILocalizationService>();  
  
 var values = await Task.WhenAll(  
 Enum.GetValues(typeof(TEnum)).OfType<TEnum>()  
 .Where(enumValue => valuesToExclude == null || !valuesToExclude.Contains(Convert.ToInt32(enumValue)))  
 .Select(async enumValue => new  
 {  
 ID = Convert.ToInt32(enumValue),  
 Name = useLocalization  
 ? await localizationService.GetResourceAsync($"Enums.{typeof(TEnum).Name}.{enumValue}")  
 : CommonHelper.SplitCamelCaseWord(enumValue.ToString())  
 })  
 );  
  
 object selectedValue = null;  
 if (markCurrentAsSelected)  
 selectedValue = Convert.ToInt32(enumObj);  
  
 return new SelectList(values, "ID", "Name", selectedValue);  
 }  
  
 /// <summary>  
 /// Convert to select list  
 /// </summary>  
 /// <typeparam name="T">Type</typeparam>  
 /// <param name="objList">List of objects</param>  
 /// <param name="selector">Selector for name</param>  
 /// <returns>SelectList</returns>  
 public static SelectList ToSelectList<T>(this T objList, Func<BaseEntity, string> selector) where T : IEnumerable<BaseEntity>  
 {  
 return new SelectList(objList.Select(p => new { ID = p.Id, Name = selector(p) }), "ID", "Name");  
 }  
  
 /// <summary>  
 /// Convert to lookup-like dictionary, for JSON serialization  
 /// </summary>  
 /// <typeparam name="T">Source type</typeparam>  
 /// <typeparam name="TKey">Key type</typeparam>  
 /// <typeparam name="TValue">Value type</typeparam>  
 /// <param name="xs">List of objects</param>  
 /// <param name="keySelector">A key-selector function</param>  
 /// <param name="valueSelector">A value-selector function</param>  
 /// <returns>A dictionary with values grouped by key</returns>  
 public static IDictionary<TKey, IList<TValue>> ToGroupedDictionary<T, TKey, TValue>(  
 this IEnumerable<T> xs,  
 Func<T, TKey> keySelector,  
 Func<T, TValue> valueSelector)  
 {  
 var result = new Dictionary<TKey, IList<TValue>>();  
  
 foreach (var x in xs)  
 {  
 var key = keySelector(x);  
 var value = valueSelector(x);  
  
 if (result.TryGetValue(key, out var list))  
 list.Add(value);  
 else  
 result[key] = new List<TValue> { value };  
 }  
  
 return result;  
 }  
  
 /// <summary>  
 /// Convert to lookup-like dictionary, for JSON serialization  
 /// </summary>  
 /// <typeparam name="T">Source type</typeparam>  
 /// <typeparam name="TKey">Key type</typeparam>  
 /// <param name="xs">List of objects</param>  
 /// <param name="keySelector">A key-selector function</param>  
 /// <returns>A dictionary with values grouped by key</returns>  
 public static IDictionary<TKey, IList<T>> ToGroupedDictionary<T, TKey>(  
 this IEnumerable<T> xs,  
 Func<T, TKey> keySelector)  
 {  
 return xs.ToGroupedDictionary(keySelector, x => x);  
 }  
}

## AuditTrailService.cs

using System;  
using System.Threading.Tasks;  
using App.Core.Domain.Audit;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Audit  
{  
 public class AuditTrailService : BaseService, IAuditTrailService  
 {  
 private readonly IRepository<AuditTrail> \_auditRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public AuditTrailService(IRepository<AuditTrail> auditRepository, ILocalizationService localizationService)  
 {  
 \_auditRepository = auditRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult> LogAsync(AuditTrail entry)  
 {  
 if (entry == null)  
 return Failed(await \_localizationService.GetResourceAsync("Audit.Log.Null"));  
  
 if (entry.ChangedOnUtc == default)  
 entry.ChangedOnUtc = DateTime.UtcNow;  
  
 await \_auditRepository.InsertAsync(entry);  
 return Success(await \_localizationService.GetResourceAsync("Audit.Log.Success"));  
 }  
  
 public async Task<ServiceResult> LogCreateAsync(string entityName, int entityId, int userId, string comment = null)  
 {  
 var entry = new AuditTrail  
 {  
 EntityName = entityName,  
 EntityId = entityId,  
 Action = AuditActionType.Create,  
 ActionId = (int)AuditActionType.Create,  
 ChangedByUserId = userId,  
 ChangedOnUtc = DateTime.UtcNow,  
 Comment = comment  
 };  
  
 return await LogAsync(entry);  
 }  
  
 public async Task<ServiceResult> LogUpdateAsync(string entityName, int entityId, int userId, string field = null, string oldValue = null, string newValue = null, string comment = null)  
 {  
 var entry = new AuditTrail  
 {  
 EntityName = entityName,  
 EntityId = entityId,  
 Action = AuditActionType.Update,  
 ActionId = (int)AuditActionType.Update,  
 ChangedByUserId = userId,  
 ChangedOnUtc = DateTime.UtcNow,  
 FieldName = field,  
 OldValue = oldValue,  
 NewValue = newValue,  
 Comment = comment  
 };  
  
 return await LogAsync(entry);  
 }  
  
 public async Task<ServiceResult> LogDeleteAsync(string entityName, int entityId, int userId, string comment = null)  
 {  
 var entry = new AuditTrail  
 {  
 EntityName = entityName,  
 EntityId = entityId,  
 Action = AuditActionType.Delete,  
 ActionId = (int)AuditActionType.Delete,  
 ChangedByUserId = userId,  
 ChangedOnUtc = DateTime.UtcNow,  
 Comment = comment  
 };  
  
 return await LogAsync(entry);  
 }  
  
 public async Task<PagedResult<AuditTrail>> SearchAsync(  
 DateTime? fromUtc = null,  
 DateTime? toUtc = null,  
 int? changedByUserId = null,  
 string entityName = null,  
 int? entityId = null,  
 string fieldName = null,  
 AuditActionType? action = null,  
 int pageIndex = 0,  
 int pageSize = 50)  
 {  
 var query = \_auditRepository.Table;  
  
 if (fromUtc.HasValue)  
 query = query.Where(x => x.ChangedOnUtc >= fromUtc.Value);  
  
 if (toUtc.HasValue)  
 query = query.Where(x => x.ChangedOnUtc <= toUtc.Value);  
  
 if (changedByUserId.HasValue)  
 query = query.Where(x => x.ChangedByUserId == changedByUserId.Value);  
  
 if (!string.IsNullOrWhiteSpace(entityName))  
 query = query.Where(x => x.EntityName == entityName);  
  
 if (entityId.HasValue)  
 query = query.Where(x => x.EntityId == entityId.Value);  
  
 if (!string.IsNullOrWhiteSpace(fieldName))  
 query = query.Where(x => x.FieldName == fieldName);  
  
 if (action.HasValue)  
 query = query.Where(x => x.ActionId == (int)action.Value);  
  
 var totalCount = query.Count();  
  
 var items = query  
 .OrderByDescending(x => x.ChangedOnUtc)  
 .Skip(pageIndex \* pageSize)  
 .Take(pageSize)  
 .ToList();  
  
 return await Task.FromResult(new PagedResult<AuditTrail>(items, totalCount, pageIndex, pageSize));  
 }  
 }  
}

## IAuditTrailService.cs

using System.Threading.Tasks;  
using App.Core.Domain.Audit;  
using App.Services.Common;  
  
namespace App.Services.Audit  
{  
 public interface IAuditTrailService  
 {  
 Task<ServiceResult> LogAsync(AuditTrail entry);  
  
 Task<ServiceResult> LogCreateAsync(string entityName, int entityId, int userId, string comment = null);  
  
 Task<ServiceResult> LogUpdateAsync(  
 string entityName,  
 int entityId,  
 int userId,  
 string field = null,  
 string oldValue = null,  
 string newValue = null,  
 string comment = null);  
  
 Task<ServiceResult> LogDeleteAsync(string entityName, int entityId, int userId, string comment = null);  
  
 Task<PagedResult<AuditTrail>> SearchAsync(  
 DateTime? fromUtc = null,  
 DateTime? toUtc = null,  
 int? changedByUserId = null,  
 string entityName = null,  
 int? entityId = null,  
 string fieldName = null,  
 AuditActionType? action = null,  
 int pageIndex = 0,  
 int pageSize = 50);  
 }  
}

## CalendarService.cs

using System;  
using System.Linq;  
using System.Threading.Tasks;  
using System.Collections.Generic;  
using App.Core.Domain.Directory;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Calendar  
{  
 public class CalendarService : BaseService, ICalendarService  
 {  
 private readonly IRepository<CalendarEvent> \_calendarEventRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public CalendarService(IRepository<CalendarEvent> calendarEventRepository, ILocalizationService localizationService)  
 {  
 \_calendarEventRepository = calendarEventRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult<CalendarEvent>> GetAsync(int id)  
 {  
 var e = await \_calendarEventRepository.GetByIdAsync(id);  
 if (e == null) return ServiceResult<CalendarEvent>.Failed(await \_localizationService.GetResourceAsync("Calendar.Event.NotFound"));  
 return ServiceResult<CalendarEvent>.Success(e);  
 }  
  
 public async Task<ServiceResult<IReadOnlyList<CalendarEvent>>> GetRangeAsync(DateTime fromUtc, DateTime toUtc)  
 {  
 var list = await \_calendarEventRepository.GetAllAsync(q => q.Where(x => x.StartDateUtc <= toUtc && x.EndDateUtc >= fromUtc));  
 return ServiceResult<IReadOnlyList<CalendarEvent>>.Success(list.ToList());  
 }  
  
 public async Task<ServiceResult<CalendarEvent>> CreateAsync(CalendarEvent model)  
 {  
 if (model == null) return ServiceResult<CalendarEvent>.Failed(await \_localizationService.GetResourceAsync("Errors.NullModel"));  
 if (model.EndDateUtc < model.StartDateUtc) return ServiceResult<CalendarEvent>.Failed(await \_localizationService.GetResourceAsync("Calendar.InvalidRange", "Invalid range"));  
 await \_calendarEventRepository.InsertAsync(model);  
 return ServiceResult<CalendarEvent>.Success(model);  
 }  
  
 public async Task<ServiceResult> UpdateAsync(int id, CalendarEvent model)  
 {  
 var e = await \_calendarEventRepository.GetByIdAsync(id);  
 if (e == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Calendar.Event.NotFound"));  
 e.Title = model.Title;  
 e.Description = model.Description;  
 e.StartDateUtc = model.StartDateUtc;  
 e.EndDateUtc = model.EndDateUtc;  
 e.IsHoliday = model.IsHoliday;  
 e.EventType = model.EventType;  
 await \_calendarEventRepository.UpdateAsync(e);  
 return ServiceResult.SuccessResult();  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var e = await \_calendarEventRepository.GetByIdAsync(id);  
 if (e == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Calendar.Event.NotFound"));  
 await \_calendarEventRepository.DeleteAsync(e);  
 return ServiceResult.SuccessResult();  
 }  
 }  
}

## ICalendarService.cs

using System;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Directory;  
using App.Services.Common;  
  
namespace App.Services.Calendar  
{  
 public interface ICalendarService  
 {  
 Task<ServiceResult<CalendarEvent>> GetAsync(int id);  
 Task<ServiceResult<IReadOnlyList<CalendarEvent>>> GetRangeAsync(DateTime fromUtc, DateTime toUtc);  
 Task<ServiceResult<CalendarEvent>> CreateAsync(CalendarEvent model);  
 Task<ServiceResult> UpdateAsync(int id, CalendarEvent model);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## BaseService.cs

﻿using System.Collections.Generic;  
  
namespace App.Services.Common  
{  
 /// <summary>  
 /// Provides common helpers for service implementations  
 /// </summary>  
 public abstract class BaseService  
 {  
 #region Non-generic results  
  
 protected ServiceResult Success(string message = null)  
 => ServiceResult.SuccessResult(message);  
  
 protected ServiceResult Failed(string error)  
 => ServiceResult.Failed(error);  
  
 protected ServiceResult Failed(IDictionary<string, string> errors)  
 => ServiceResult.Failed(errors);  
  
 #endregion  
  
 #region Generic results  
  
 protected ServiceResult<T> Success<T>(T data, string message = null)  
 => ServiceResult<T>.Success(data, message);  
  
 protected ServiceResult<T> Failed<T>(string error)  
 => ServiceResult<T>.Failed(error);  
  
 protected ServiceResult<T> Failed<T>(IDictionary<string, string> errors)  
 => ServiceResult<T>.Failed(errors);  
  
 #endregion  
 }  
}

## Guards.cs

using System;  
using System.Threading.Tasks;  
  
namespace App.Services.Common  
{  
 public static class Guards  
 {  
 public static void NotNull(object obj, string paramName, Func<string> localizedMessageFactory = null)  
 {  
 if (obj is null)  
 throw new ArgumentNullException(paramName, localizedMessageFactory?.Invoke());  
 }  
  
 public static Task<T> NullToDefaultAsync<T>(Task<T> task, T defaultValue = default) => task ?? Task.FromResult(defaultValue);  
 }  
}

## PagedResult.cs

﻿using System.Collections.Generic;  
  
namespace App.Services.Common  
{  
 public class PagedResult<T>  
 {  
 public IList<T> Items { get; }  
 public int TotalCount { get; }  
 public int PageIndex { get; }  
 public int PageSize { get; }  
  
 public PagedResult(IList<T> items, int totalCount, int pageIndex, int pageSize)  
 {  
 Items = items;  
 TotalCount = totalCount;  
 PageIndex = pageIndex;  
 PageSize = pageSize;  
 }  
 }  
}

## ServiceResult.cs

using System.Collections.Generic;  
  
namespace App.Services.Common  
{  
 public class ServiceResult  
 {  
 public bool Success { get; private set; }  
 public string Message { get; private set; }  
 public string Error { get; private set; }  
 public IDictionary<string, string> Errors { get; private set; }  
  
 protected ServiceResult() { }  
  
 protected ServiceResult(bool success, string message = null, string error = null, IDictionary<string, string> errors = null)  
 {  
 Success = success;  
 Message = message;  
 Error = error;  
 Errors = errors;  
 }  
  
 #region Factory methods  
  
 public static ServiceResult SuccessResult(string message = null)  
 => new ServiceResult(true, message);  
  
 public static ServiceResult Failed(string error)  
 => new ServiceResult(false, null, error);  
  
 public static ServiceResult Failed(IDictionary<string, string> errors)  
 => new ServiceResult(false, null, null, errors);  
  
 #endregion  
 }  
  
 public class ServiceResult<T> : ServiceResult  
 {  
 public T Data { get; private set; }  
  
 private ServiceResult(T data, string message = null)  
 : base(true, message)  
 {  
 Data = data;  
 }  
  
 private ServiceResult(string error)  
 : base(false, null, error) { }  
  
 private ServiceResult(IDictionary<string, string> errors)  
 : base(false, null, null, errors) { }  
  
 #region Factory methods  
  
 public static ServiceResult<T> Success(T data, string message = null)  
 => new ServiceResult<T>(data, message);  
  
 public static new ServiceResult<T> Failed(string error)  
 => new ServiceResult<T>(error);  
  
 public static new ServiceResult<T> Failed(IDictionary<string, string> errors)  
 => new ServiceResult<T>(errors);  
  
 #endregion  
 }  
}

## ILanguageService.cs

using App.Core.Domain.Localization;  
  
public interface ILanguageService  
{  
 // existing  
 Task<Language> GetByIdAsync(int id);  
 Task<IList<Language>> GetAllAsync(bool onlyPublished = true);  
 Task InsertAsync(Language language);  
 Task UpdateAsync(Language language);  
 Task DeleteAsync(Language language);  
  
 // new  
 Task<Language> GetCurrentLanguageAsync();  
 Task SetCurrentLanguageAsync(int languageId);  
}

## ILocalizationService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Localization;  
  
namespace App.Services.Localization  
{  
 public interface ILocalizationService  
 {  
 // Resources  
 Task<string> GetResourceAsync(string key, string defaultValue = null);  
 Task<LocaleStringResource> GetByIdAsync(int id);  
 Task<LocaleStringResource> GetByNameAsync(string resourceName, int languageId);  
 Task<IList<LocaleStringResource>> GetAllResourcesAsync(int languageId);  
 Task InsertResourceAsync(LocaleStringResource resource);  
 Task UpdateResourceAsync(LocaleStringResource resource);  
 Task DeleteResourceAsync(LocaleStringResource resource);  
  
 // Helpers  
 Task<string> GetLocalizedEnumAsync<TEnum>(TEnum enumValue) where TEnum : struct;  
 string FormatMessage(string template, params object[] args);  
 }  
}

## LanguageService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Localization;  
using App.Core.RepositoryServices;  
  
namespace App.Services.Localization  
{  
 public class LanguageService : ILanguageService  
 {  
 private readonly IRepository<Language> \_languageRepository;  
 private static int \_currentLanguageId = 1; // default = Arabic/English as per seed  
  
 public LanguageService(IRepository<Language> languageRepository)  
 {  
 \_languageRepository = languageRepository;  
 }  
  
 public async Task<Language> GetByIdAsync(int id)  
 => await \_languageRepository.GetByIdAsync(id);  
  
 public async Task<IList<Language>> GetAllAsync(bool onlyPublished = true)  
 {  
 return await \_languageRepository.GetAllAsync(q =>  
 {  
 if (onlyPublished)  
 q = q.Where(l => l.Published);  
 return q.OrderBy(l => l.DisplayOrder);  
 });  
 }  
  
 public async Task InsertAsync(Language language)  
 => await \_languageRepository.InsertAsync(language);  
  
 public async Task UpdateAsync(Language language)  
 => await \_languageRepository.UpdateAsync(language);  
  
 public async Task DeleteAsync(Language language)  
 => await \_languageRepository.DeleteAsync(language);  
  
 // new  
 public async Task<Language> GetCurrentLanguageAsync()  
 => await GetByIdAsync(\_currentLanguageId);  
  
 public async Task SetCurrentLanguageAsync(int languageId)  
 {  
 \_currentLanguageId = languageId;  
 await Task.CompletedTask;  
 }  
 }  
}

## LocalizationService.cs

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Localization;  
using App.Core.RepositoryServices;  
  
namespace App.Services.Localization  
{  
 public class LocalizationService : ILocalizationService  
 {  
 private readonly IRepository<LocaleStringResource> \_resourceRepository;  
 private readonly ILanguageService \_languageService;  
  
 public LocalizationService(  
 IRepository<LocaleStringResource> resourceRepository,  
 ILanguageService languageService)  
 {  
 \_resourceRepository = resourceRepository;  
 \_languageService = languageService;  
 }  
  
 public async Task<string> GetResourceAsync(string key, string defaultValue = null)  
 {  
 var language = await \_languageService.GetCurrentLanguageAsync();  
 var resource = await \_resourceRepository.GetAllAsync(q =>  
 q.Where(r => r.ResourceName == key && r.LanguageId == language.Id));  
  
 var value = resource.FirstOrDefault()?.ResourceValue;  
 return string.IsNullOrEmpty(value) ? defaultValue ?? key : value;  
 }  
  
 public async Task<LocaleStringResource> GetByIdAsync(int id)  
 => await \_resourceRepository.GetByIdAsync(id);  
  
 public async Task<LocaleStringResource> GetByNameAsync(string resourceName, int languageId)  
 {  
 var list = await \_resourceRepository.GetAllAsync(q =>  
 q.Where(r => r.ResourceName == resourceName && r.LanguageId == languageId));  
 return list.FirstOrDefault();  
 }  
  
 public async Task<IList<LocaleStringResource>> GetAllResourcesAsync(int languageId)  
 {  
 return await \_resourceRepository.GetAllAsync(q =>  
 q.Where(r => r.LanguageId == languageId));  
 }  
  
 public async Task InsertResourceAsync(LocaleStringResource resource)  
 => await \_resourceRepository.InsertAsync(resource);  
  
 public async Task UpdateResourceAsync(LocaleStringResource resource)  
 => await \_resourceRepository.UpdateAsync(resource);  
  
 public async Task DeleteResourceAsync(LocaleStringResource resource)  
 => await \_resourceRepository.DeleteAsync(resource);  
  
 // New: Localize Enums  
 public async Task<string> GetLocalizedEnumAsync<TEnum>(TEnum enumValue) where TEnum : struct  
 {  
 var enumName = enumValue.ToString();  
 var key = $"Enums.{typeof(TEnum).Name}.{enumName}";  
 // e.g., Enums.RegistrationStatus.Approved  
  
 var result = await GetResourceAsync(key, enumName);  
 return result ?? enumName;  
 }  
  
 // New: Format message with placeholders  
 public string FormatMessage(string template, params object[] args)  
 {  
 if (string.IsNullOrEmpty(template))  
 return string.Empty;  
  
 return string.Format(template, args);  
 }  
 }  
}

## CorrespondenceService.cs

using System.Linq;  
using System.Threading.Tasks;  
using System.Collections.Generic;  
using App.Core.Domain.Correspondences;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Correspondences  
{  
 public class CorrespondenceService : BaseService, ICorrespondenceService  
 {  
 private readonly IRepository<Correspondence> \_correspondenceRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public CorrespondenceService(IRepository<Correspondence> correspondenceRepository, ILocalizationService localizationService)  
 {  
 \_correspondenceRepository = correspondenceRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult<Correspondence>> GetAsync(int id)  
 {  
 var entity = await \_correspondenceRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult<Correspondence>.Failed(await \_localizationService.GetResourceAsync("Correspondence.NotFound"));  
 return ServiceResult<Correspondence>.Success(entity);  
 }  
  
 public async Task<ServiceResult<IReadOnlyList<Correspondence>>> InboxAsync(int userId)  
 {  
 var list = await \_correspondenceRepository.GetAllAsync(q => q.Where(x => x.RecipientUserId == userId));  
 return ServiceResult<IReadOnlyList<Correspondence>>.Success(list.ToList());  
 }  
  
 public async Task<ServiceResult<IReadOnlyList<Correspondence>>> OutboxAsync(int userId)  
 {  
 var list = await \_correspondenceRepository.GetAllAsync(q => q.Where(x => x.SenderUserId == userId));  
 return ServiceResult<IReadOnlyList<Correspondence>>.Success(list.ToList());  
 }  
  
 public async Task<ServiceResult<Correspondence>> SendAsync(Correspondence model)  
 {  
 if (model == null) return ServiceResult<Correspondence>.Failed(await \_localizationService.GetResourceAsync("Errors.NullModel"));  
 await \_correspondenceRepository.InsertAsync(model);  
 return ServiceResult<Correspondence>.Success(model);  
 }  
  
 public async Task<ServiceResult> MarkAsAsync(int id, string status)  
 {  
 var entity = await \_correspondenceRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Correspondence.NotFound"));  
 entity.Status = status;  
 await \_correspondenceRepository.UpdateAsync(entity);  
 return ServiceResult.SuccessResult();  
 }  
 }  
}

## ICorrespondenceService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Correspondences;  
using App.Services.Common;  
  
namespace App.Services.Correspondences  
{  
 public interface ICorrespondenceService  
 {  
 Task<ServiceResult<Correspondence>> GetAsync(int id);  
 Task<ServiceResult<IReadOnlyList<Correspondence>>> InboxAsync(int userId);  
 Task<ServiceResult<IReadOnlyList<Correspondence>>> OutboxAsync(int userId);  
 Task<ServiceResult<Correspondence>> SendAsync(Correspondence model);  
 Task<ServiceResult> MarkAsAsync(int id, string status);  
 }  
}

## CountryService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Directory;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Directory  
{  
 public class CountryService : BaseService, ICountryService  
 {  
 private readonly IRepository<Country> \_countryRepository;  
 private readonly IRepository<StateProvince> \_stateRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public CountryService(  
 IRepository<Country> countryRepository,  
 IRepository<StateProvince> stateRepository,  
 ILocalizationService localizationService)  
 {  
 \_countryRepository = countryRepository;  
 \_stateRepository = stateRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<Country> GetByIdAsync(int id)  
 => await \_countryRepository.GetByIdAsync(id);  
  
 public async Task<IList<Country>> GetAllAsync(bool onlyActive = true)  
 {  
 var list = await \_countryRepository.GetAllAsync(q =>  
 onlyActive ? q.Where(c => c.Published) : q);  
 return list.OrderBy(c => c.DisplayOrder).ToList();  
 }  
  
 public async Task<ServiceResult<Country>> InsertAsync(Country country)  
 {  
 if (country == null)  
 return Failed<Country>(await \_localizationService.GetResourceAsync("Country.Insert.Null"));  
  
 if (await ExistsAsync(country.TwoLetterIsoCode))  
 return Failed<Country>(await \_localizationService.GetResourceAsync("Country.Insert.Duplicate"));  
  
 await \_countryRepository.InsertAsync(country);  
 return Success(country, await \_localizationService.GetResourceAsync("Country.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<Country>> UpdateAsync(Country country)  
 {  
 if (country == null)  
 return Failed<Country>(await \_localizationService.GetResourceAsync("Country.Update.Null"));  
  
 await \_countryRepository.UpdateAsync(country);  
 return Success(country, await \_localizationService.GetResourceAsync("Country.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var country = await \_countryRepository.GetByIdAsync(id);  
 if (country == null)  
 return Failed(await \_localizationService.GetResourceAsync("Country.NotFound"));  
  
 await \_countryRepository.DeleteAsync(country);  
 return Success(await \_localizationService.GetResourceAsync("Country.Delete.Success"));  
 }  
  
 public async Task<Country> GetByCodeAsync(string twoLetterIsoCode)  
 {  
 var countries = await \_countryRepository.GetAllAsync(q => q.Where(c => c.TwoLetterIsoCode == twoLetterIsoCode));  
 return countries.FirstOrDefault();  
 }  
  
 public async Task<Country> GetByNameAsync(string name)  
 {  
 var countries = await \_countryRepository.GetAllAsync(q => q.Where(c => c.Name == name));  
 return countries.FirstOrDefault();  
 }  
  
 public async Task<PagedResult<Country>> SearchAsync(string keyword = null, bool? isActive = null, int pageIndex = 0, int pageSize = int.MaxValue)  
 {  
 var query = \_countryRepository.Table;  
  
 if (!string.IsNullOrEmpty(keyword))  
 query = query.Where(c => c.Name.Contains(keyword) || c.TwoLetterIsoCode.Contains(keyword));  
  
 if (isActive.HasValue)  
 query = query.Where(c => c.Published == isActive.Value);  
  
 var totalCount = query.Count();  
  
 var items = query  
 .OrderBy(c => c.DisplayOrder)  
 .Skip(pageIndex \* pageSize)  
 .Take(pageSize)  
 .ToList();  
  
 return await Task.FromResult(new PagedResult<Country>(items, totalCount, pageIndex, pageSize));  
 }  
  
 public async Task<IList<StateProvince>> GetStatesByCountryIdAsync(int countryId)  
 {  
 var states = await \_stateRepository.GetAllAsync(q => q.Where(s => s.CountryId == countryId));  
 return states.OrderBy(s => s.DisplayOrder).ToList();  
 }  
  
 public async Task<bool> ExistsAsync(string twoLetterIsoCode)  
 {  
 var exists = await \_countryRepository.GetAllAsync(q => q.Where(c => c.TwoLetterIsoCode == twoLetterIsoCode));  
 return exists.Any();  
 }  
 }  
}

## ICountryService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Directory;  
using App.Services.Common;  
  
namespace App.Services.Directory  
{  
 public interface ICountryService  
 {  
 // CRUD  
 Task<Country> GetByIdAsync(int id);  
 Task<IList<Country>> GetAllAsync(bool onlyActive = true);  
 Task<ServiceResult<Country>> InsertAsync(Country country);  
 Task<ServiceResult<Country>> UpdateAsync(Country country);  
 Task<ServiceResult> DeleteAsync(int id);  
  
 // Lookup  
 Task<Country> GetByCodeAsync(string twoLetterIsoCode);  
 Task<Country> GetByNameAsync(string name);  
  
 // Search & Paging  
 Task<PagedResult<Country>> SearchAsync(  
 string keyword = null,  
 bool? isActive = null,  
 int pageIndex = 0,  
 int pageSize = int.MaxValue);  
  
 // Hierarchy  
 Task<IList<StateProvince>> GetStatesByCountryIdAsync(int countryId);  
  
 // Utility  
 Task<bool> ExistsAsync(string twoLetterIsoCode);  
 }  
}

## IStateProvinceService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Directory;  
using App.Services.Common;  
  
namespace App.Services.Directory  
{  
 public interface IStateProvinceService  
 {  
 Task<StateProvince> GetByIdAsync(int id);  
 Task<IList<StateProvince>> GetByCountryIdAsync(int countryId);  
 Task<ServiceResult<StateProvince>> InsertAsync(StateProvince state);  
 Task<ServiceResult<StateProvince>> UpdateAsync(StateProvince state);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## StateProvinceService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Directory;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Directory  
{  
 public class StateProvinceService : BaseService, IStateProvinceService  
 {  
 private readonly IRepository<StateProvince> \_stateRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public StateProvinceService(IRepository<StateProvince> stateRepository, ILocalizationService localizationService)  
 {  
 \_stateRepository = stateRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<StateProvince> GetByIdAsync(int id)  
 => await \_stateRepository.GetByIdAsync(id);  
  
 public async Task<IList<StateProvince>> GetByCountryIdAsync(int countryId)  
 {  
 var states = await \_stateRepository.GetAllAsync(q => q.Where(s => s.CountryId == countryId));  
 return states.OrderBy(s => s.DisplayOrder).ToList();  
 }  
  
 public async Task<ServiceResult<StateProvince>> InsertAsync(StateProvince state)  
 {  
 if (state == null)  
 return Failed<StateProvince>(await \_localizationService.GetResourceAsync("State.Insert.Null"));  
  
 await \_stateRepository.InsertAsync(state);  
 return Success(state, await \_localizationService.GetResourceAsync("State.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<StateProvince>> UpdateAsync(StateProvince state)  
 {  
 if (state == null)  
 return Failed<StateProvince>(await \_localizationService.GetResourceAsync("State.Update.Null"));  
  
 await \_stateRepository.UpdateAsync(state);  
 return Success(state, await \_localizationService.GetResourceAsync("State.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var state = await \_stateRepository.GetByIdAsync(id);  
 if (state == null)  
 return Failed(await \_localizationService.GetResourceAsync("State.NotFound"));  
  
 await \_stateRepository.DeleteAsync(state);  
 return Success(await \_localizationService.GetResourceAsync("State.Delete.Success"));  
 }  
 }  
}

## FileValidationService.cs

using System;  
using System.IO;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Files  
{  
 public class FileValidationService : IFileValidationService  
 {  
 private long \_maxSizeBytes = 10 \* 1024 \* 1024; // 10 MB default  
 private string[] \_allowedExtensions = new[] { ".pdf", ".doc", ".docx", ".png", ".jpg", ".jpeg" };  
  
 public void Configure(long maxSizeBytes, string[] allowedExtensions)  
 {  
 if (maxSizeBytes > 0) \_maxSizeBytes = maxSizeBytes;  
 if (allowedExtensions != null && allowedExtensions.Length > 0)  
 \_allowedExtensions = allowedExtensions.Select(x => x.ToLowerInvariant()).ToArray();  
 }  
  
 public Task<bool> ValidateAsync(string fileName, long contentLengthBytes)  
 {  
 if (string.IsNullOrWhiteSpace(fileName)) return Task.FromResult(false);  
 var ext = Path.GetExtension(fileName)?.ToLowerInvariant();  
 if (string.IsNullOrWhiteSpace(ext)) return Task.FromResult(false);  
 if (!\_allowedExtensions.Contains(ext)) return Task.FromResult(false);  
 if (contentLengthBytes <= 0 || contentLengthBytes > \_maxSizeBytes) return Task.FromResult(false);  
 return Task.FromResult(true);  
 }  
 }  
}

## IFileValidationService.cs

using System.Threading.Tasks;  
  
namespace App.Services.Files  
{  
 public interface IFileValidationService  
 {  
 Task<bool> ValidateAsync(string fileName, long contentLengthBytes);  
 void Configure(long maxSizeBytes, string[] allowedExtensions);  
 }  
}

## GenericAttributeService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Common;  
using App.Core.RepositoryServices;  
  
namespace App.Services.Common  
{  
 public class GenericAttributeService : IGenericAttributeService  
 {  
 private readonly IRepository<GenericAttribute> \_attributeRepository;  
  
 public GenericAttributeService(IRepository<GenericAttribute> attributeRepository)  
 {  
 \_attributeRepository = attributeRepository;  
 }  
  
 public async Task<IList<GenericAttribute>> GetAttributesForEntityAsync(int entityId, string entityName)  
 {  
 return (await \_attributeRepository.GetAllAsync(q =>  
 q.Where(a => a.EntityId == entityId && a.KeyGroup == entityName))).ToList();  
 }  
  
 public async Task<GenericAttribute> GetAttributeAsync(int entityId, string entityName, string key)  
 {  
 return (await \_attributeRepository.GetAllAsync(q =>  
 q.Where(a => a.EntityId == entityId && a.KeyGroup == entityName && a.Key == key)))  
 .FirstOrDefault();  
 }  
  
 public async Task SetAttributeAsync<T>(int entityId, string entityName, string key, T value)  
 {  
 var existing = await GetAttributeAsync(entityId, entityName, key);  
 var valueStr = value?.ToString();  
  
 if (existing != null)  
 {  
 existing.Value = valueStr;  
 await \_attributeRepository.UpdateAsync(existing);  
 }  
 else  
 {  
 var attr = new GenericAttribute  
 {  
 EntityId = entityId,  
 KeyGroup = entityName,  
 Key = key,  
 Value = valueStr  
 };  
 await \_attributeRepository.InsertAsync(attr);  
 }  
 }  
  
 public async Task DeleteAttributeAsync(int entityId, string entityName, string key)  
 {  
 var existing = await GetAttributeAsync(entityId, entityName, key);  
 if (existing != null)  
 await \_attributeRepository.DeleteAsync(existing);  
 }  
 }  
}

## IGenericAttributeService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Common;  
  
namespace App.Services.Common  
{  
 public interface IGenericAttributeService  
 {  
 Task<IList<GenericAttribute>> GetAttributesForEntityAsync(int entityId, string entityName);  
 Task<GenericAttribute> GetAttributeAsync(int entityId, string entityName, string key);  
 Task SetAttributeAsync<T>(int entityId, string entityName, string key, T value);  
 Task DeleteAttributeAsync(int entityId, string entityName, string key);  
 }  
}

## AppInstallationDefaults.cs

namespace App.Services.Installation  
{  
 public static class AppInstallationDefaults  
 {  
 public const string DefaultAdminEmail = "admin@system.local";  
 public const string DefaultAdminUsername = "admin";  
 public const string DefaultAdminPassword = "Admin@123";  
  
 public const string DefaultSiteName = "Supervision System";  
 // Match deployment region (Africa/Cairo)  
 public const string DefaultTimeZone = "Africa/Cairo";  
 public const string DefaultLanguage = "en-US";  
 }  
}

## IInstallationService.cs

using System.Threading.Tasks;  
  
namespace App.Services.Installation  
{  
 public interface IInstallationService  
 {  
 /// <summary>  
 /// Run schema migrations (create all tables)  
 /// </summary>  
 ///Task InstallSchemaAsync();  
  
 /// <summary>  
 /// Install required system data (languages, countries, admin user, roles, etc.)  
 /// </summary>  
 Task InstallRequiredDataAsync();  
  
 /// <summary>  
 /// Install sample/test data  
 /// </summary>  
 Task InstallSampleDataAsync();  
 }  
}

## InstallationService.cs

using App.Data.Migrations.Installation;  
using App.Data.MigratorServices;  
using System.Threading.Tasks;  
  
namespace App.Services.Installation  
{  
 public class InstallationService : IInstallationService  
 {  
 private readonly InstallRequiredData \_installRequiredData;  
 private readonly InstallSampleData \_installSampleData;  
 private readonly IMigrationManager \_migrationManager;  
  
 public InstallationService(InstallRequiredData installRequiredData, InstallSampleData installSampleData)  
 {  
 \_installRequiredData = installRequiredData;  
 \_installSampleData = installSampleData;  
 }  
  
 /// <summary>  
 /// Run schema migrations (create all tables)  
 /// </summary>  
 //public async Task InstallSchemaAsync()  
 //{  
 // // Apply all schema migrations (tables creation)  
 // \_migrationManager.ApplyUpMigrations(typeof(SchemaMigration).Assembly, MigrationProcessType.Installation, false);  
  
 // await Task.CompletedTask;  
 //}  
  
 public async Task InstallRequiredDataAsync()  
 {  
 await \_installRequiredData.InstallAsync();  
 }  
  
 public async Task InstallSampleDataAsync()  
 {  
 await \_installSampleData.InstallAsync();  
 }  
 }  
}

## InstallRequiredData.cs

using System;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.RepositoryServices;  
using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.Domain.Registrations;  
using App.Core.Domain.Settings;  
using App.Core.Domain.Common;  
using App.Services.Security;  
using App.Services.Settings;  
  
namespace App.Services.Installation  
{  
 /// <summary>  
 /// Seeds required data that must exist before the app can run.  
 /// </summary>  
 public partial class InstallRequiredData  
 {  
 private readonly IRepository<Role> \_roleRepository;  
 private readonly IRepository<User> \_userRepository;  
 private readonly IRepository<UserRole> \_userRoleRepository;  
 private readonly IRepository<WorkflowStep> \_workflowStepRepository;  
 private readonly IRepository<Template> \_templateRepository;  
 private readonly ISettingService \_settingService;  
 private readonly IEncryptionService \_encryptionService;  
  
 public InstallRequiredData(  
 IRepository<Role> roleRepository,  
 IRepository<User> userRepository,  
 IRepository<UserRole> userRoleRepository,  
 IRepository<WorkflowStep> workflowStepRepository,  
 IRepository<Template> templateRepository,  
 ISettingService settingService,  
 IEncryptionService encryptionService)  
 {  
 \_roleRepository = roleRepository;  
 \_userRepository = userRepository;  
 \_userRoleRepository = userRoleRepository;  
 \_workflowStepRepository = workflowStepRepository;  
 \_templateRepository = templateRepository;  
 \_settingService = settingService;  
 \_encryptionService = encryptionService;  
 }  
  
 public async Task InstallAsync()  
 {  
 await SeedRolesAsync();  
 await SeedAdminUserAsync();  
 await SeedWorkflowStepsAsync();  
 await SeedDefaultSettingsAsync();  
 await SeedNotificationTemplatesAsync();  
 }  
  
 #region Roles  
 private async Task SeedRolesAsync()  
 {  
 var required = new[] { "Maker", "Checker", "Regulator", "Inspector", "Admin" };  
 foreach (var roleName in required)  
 {  
 var exists = (await \_roleRepository.GetAllAsync(q => q.Where(r => r.Name == roleName))).FirstOrDefault();  
 if (exists == null)  
 {  
 var role = new Role  
 {  
 Name = roleName,  
 Description = $"System role: {roleName}",  
 IsActive = true,  
 CreatedOnUtc = DateTime.UtcNow,  
 UpdatedOnUtc = null  
 };  
 await \_roleRepository.InsertAsync(role);  
 }  
 }  
 }  
 #endregion  
  
 #region Admin user  
 private async Task SeedAdminUserAsync()  
 {  
 // Try to find admin by username  
 var admin = (await \_userRepository.GetAllAsync(q => q.Where(u => u.Username == AppInstallationDefaults.DefaultAdminUsername))).FirstOrDefault();  
 if (admin == null)  
 {  
 admin = new User  
 {  
 Username = AppInstallationDefaults.DefaultAdminUsername,  
 Email = AppInstallationDefaults.DefaultAdminEmail,  
 PasswordHash = \_encryptionService.HashPassword(AppInstallationDefaults.DefaultAdminPassword),  
 IsActive = true,  
 CreatedOnUtc = DateTime.UtcNow,  
 FailedLoginAttempts = 0  
 };  
 await \_userRepository.InsertAsync(admin);  
 }  
  
 // Ensure Admin role mapping  
 var adminRole = (await \_roleRepository.GetAllAsync(q => q.Where(r => r.Name == "Admin"))).FirstOrDefault();  
 if (adminRole != null)  
 {  
 var mappingExists = (await \_userRoleRepository.GetAllAsync(q => q.Where(m => m.UserId == admin.Id && m.RoleId == adminRole.Id))).Any();  
 if (!mappingExists)  
 {  
 await \_userRoleRepository.InsertAsync(new UserRole  
 {  
 UserId = admin.Id,  
 RoleId = adminRole.Id,  
 CreatedOnUtc = DateTime.UtcNow  
 });  
 }  
 }  
 }  
 #endregion  
  
 #region Workflow  
 private async Task SeedWorkflowStepsAsync()  
 {  
 // Draft -> Submitted -> Under Review -> Approved  
 var draft = await EnsureWorkflowStepAsync("Draft", "Maker");  
 var submitted = await EnsureWorkflowStepAsync("Submitted", "Checker");  
 var underReview = await EnsureWorkflowStepAsync("Under Review", "Regulator");  
 var approved = await EnsureWorkflowStepAsync("Approved", "Regulator");  
 var rejected = await EnsureWorkflowStepAsync("Rejected", "Regulator");  
 var returned = await EnsureWorkflowStepAsync("Returned for Edit", "Checker,Regulator");  
  
 // Link forward path  
 await LinkNextAsync(draft, submitted);  
 await LinkNextAsync(submitted, underReview);  
 await LinkNextAsync(underReview, approved);  
 // rejected/returned are terminal steps for now  
 }  
  
 private async Task<WorkflowStep> EnsureWorkflowStepAsync(string name, string roleAllowed)  
 {  
 var step = (await \_workflowStepRepository.GetAllAsync(q => q.Where(s => s.Name == name))).FirstOrDefault();  
 if (step == null)  
 {  
 step = new WorkflowStep { Name = name, RoleAllowed = roleAllowed, NextStepId = null };  
 await \_workflowStepRepository.InsertAsync(step);  
 }  
 else if (step.RoleAllowed != roleAllowed)  
 {  
 step.RoleAllowed = roleAllowed;  
 await \_workflowStepRepository.UpdateAsync(step);  
 }  
 return step;  
 }  
  
 private async Task LinkNextAsync(WorkflowStep from, WorkflowStep to)  
 {  
 if (from == null || to == null) return;  
 if (from.NextStepId != to.Id)  
 {  
 from.NextStepId = to.Id;  
 await \_workflowStepRepository.UpdateAsync(from);  
 }  
 }  
 #endregion  
  
 #region Default settings  
 private async Task SeedDefaultSettingsAsync()  
 {  
 await \_settingService.SetAsync("Site.Name", AppInstallationDefaults.DefaultSiteName);  
 await \_settingService.SetAsync("Site.TimeZone", AppInstallationDefaults.DefaultTimeZone);  
 await \_settingService.SetAsync("Site.DefaultLanguage", AppInstallationDefaults.DefaultLanguage);  
  
 await \_settingService.SetAsync("Security.Password.MinLength", 8);  
 await \_settingService.SetAsync("Security.Password.RequireDigit", true);  
 await \_settingService.SetAsync("Security.Password.RequireUppercase", true);  
 await \_settingService.SetAsync("Security.Lockout.MaxFailedAccessAttempts", 5);  
 await \_settingService.SetAsync("Security.Lockout.DefaultLockoutMinutes", 15);  
 }  
 #endregion  
  
 #region Notification templates  
 private async Task SeedNotificationTemplatesAsync()  
 {  
 await EnsureTemplateAsync("Notification.RegistrationSubmitted", "Email",  
 "Subject: Registration Submitted\nBody: Your registration has been submitted successfully.");  
 await EnsureTemplateAsync("Notification.RegistrationApproved", "Email",  
 "Subject: Registration Approved\nBody: Your registration has been approved.");  
 await EnsureTemplateAsync("Notification.RegistrationRejected", "Email",  
 "Subject: Registration Rejected\nBody: Your registration has been rejected.");  
 await EnsureTemplateAsync("Notification.RegistrationReturned", "Email",  
 "Subject: Registration Returned\nBody: Your registration has been returned for corrections.");  
 }  
  
 private async Task EnsureTemplateAsync(string name, string type, string content)  
 {  
 var existing = (await \_templateRepository.GetAllAsync(q => q.Where(t => t.Name == name && t.TemplateType == type))).FirstOrDefault();  
 if (existing == null)  
 {  
 var tpl = new Template  
 {  
 Name = name,  
 TemplateType = type,  
 Content = content,  
 CreatedByUserId = 0,  
 CreatedOnUtc = DateTime.UtcNow  
 };  
 await \_templateRepository.InsertAsync(tpl);  
 }  
 }  
 #endregion  
 }  
}

## InstallSampleData.cs

using System;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.RepositoryServices;  
using App.Core.Domain.Localization;  
using App.Core.Domain.Directory;  
using App.Core.Domain.Users;  
using App.Core.Domain.Security;  
using App.Core.Domain.Institutions;  
using App.Core.Domain.Registrations;  
using App.Services.Security;  
using App.Core.Domain.Ref;  
  
namespace App.Services.Installation  
{  
 /// <summary>  
 /// Seeds optional sample data to help verify the system after installation.  
 /// </summary>  
 public partial class InstallSampleData  
 {  
 private readonly IRepository<Language> \_languageRepository;  
 private readonly IRepository<Country> \_countryRepository;  
 private readonly IRepository<User> \_userRepository;  
 private readonly IRepository<Role> \_roleRepository;  
 private readonly IRepository<UserRole> \_userRoleRepository;  
 private readonly IRepository<Institution> \_institutionRepository;  
 private readonly IRepository<Registration> \_registrationRepository;  
 private readonly IEncryptionService \_encryptionService;  
  
 public InstallSampleData(  
 IRepository<Language> languageRepository,  
 IRepository<Country> countryRepository,  
 IRepository<User> userRepository,  
 IRepository<Role> roleRepository,  
 IRepository<UserRole> userRoleRepository,  
 IRepository<Institution> institutionRepository,  
 IRepository<Registration> registrationRepository,  
 IEncryptionService encryptionService)  
 {  
 \_languageRepository = languageRepository;  
 \_countryRepository = countryRepository;  
 \_userRepository = userRepository;  
 \_roleRepository = roleRepository;  
 \_userRoleRepository = userRoleRepository;  
 \_institutionRepository = institutionRepository;  
 \_registrationRepository = registrationRepository;  
 \_encryptionService = encryptionService;  
 }  
  
 public async Task InstallAsync()  
 {  
 await SeedCountriesAsync();  
 await SeedLanguagesAsync();  
 await SeedSampleUsersAsync();  
 await SeedSampleInstitutionAndRegistrationAsync();  
 }  
  
 private async Task SeedLanguagesAsync()  
 {  
 // English  
 var en = (await \_languageRepository.GetAllAsync(q => q.Where(l => l.UniqueSeoCode == "en"))).FirstOrDefault();  
 if (en == null)  
 {  
 en = new Language  
 {  
 Name = "English",  
 LanguageCulture = "en-US",  
 UniqueSeoCode = "en",  
 FlagImageFileName = "us.png",  
 Published = true,  
 DisplayOrder = 1,  
 Rtl = false  
 };  
 await \_languageRepository.InsertAsync(en);  
 }  
  
 // Arabic  
 var ar = (await \_languageRepository.GetAllAsync(q => q.Where(l => l.UniqueSeoCode == "ar"))).FirstOrDefault();  
 if (ar == null)  
 {  
 ar = new Language  
 {  
 Name = "Arabic",  
 LanguageCulture = "ar-SA",  
 UniqueSeoCode = "ar",  
 FlagImageFileName = "sa.png",  
 Published = true,  
 DisplayOrder = 2,  
 Rtl = true  
 };  
 await \_languageRepository.InsertAsync(ar);  
 }  
 }  
  
 private async Task SeedCountriesAsync()  
 {  
 // Minimal subset using ISO3166 helper  
 foreach (var c in ISO3166.GetCollection()) // don't flood DB in sample  
 {  
 var exists = (await \_countryRepository.GetAllAsync(q => q.Where(x => x.ThreeLetterIsoCode == c.Alpha3))).Any();  
 if (!exists)  
 {  
 var country = new Country  
 {  
 Name = c.Name,  
 TwoLetterIsoCode = c.Alpha2,  
 ThreeLetterIsoCode = c.Alpha3,  
 NumericIsoCode = c.NumericCode,  
 Published = true,  
 DisplayOrder = 1  
 };  
 await \_countryRepository.InsertAsync(country);  
 }  
 }  
 }  
  
 private async Task SeedSampleUsersAsync()  
 {  
 // Helper to ensure user + role mapping  
 async Task EnsureUserAsync(string username, string email, string password, string roleName)  
 {  
 var user = (await \_userRepository.GetAllAsync(q => q.Where(u => u.Username == username))).FirstOrDefault();  
 if (user == null)  
 {  
 user = new User  
 {  
 Username = username,  
 Email = email,  
 PasswordHash = \_encryptionService.HashPassword(password),  
 IsActive = true,  
 CreatedOnUtc = DateTime.UtcNow  
 };  
 await \_userRepository.InsertAsync(user);  
 }  
 var role = (await \_roleRepository.GetAllAsync(q => q.Where(r => r.Name == roleName))).FirstOrDefault();  
 if (role != null)  
 {  
 var mapped = (await \_userRoleRepository.GetAllAsync(q => q.Where(m => m.UserId == user.Id && m.RoleId == role.Id))).Any();  
 if (!mapped)  
 {  
 await \_userRoleRepository.InsertAsync(new UserRole  
 {  
 UserId = user.Id,  
 RoleId = role.Id,  
 CreatedOnUtc = DateTime.UtcNow  
 });  
 }  
 }  
 }  
  
 await EnsureUserAsync("maker1", "maker1@test.local", "Maker@123", "Maker");  
 await EnsureUserAsync("checker1", "checker1@test.local", "Checker@123", "Checker");  
 await EnsureUserAsync("regulator1", "regulator1@test.local", "Regulator@123", "Regulator");  
 await EnsureUserAsync("inspector1", "inspector1@test.local", "Inspector@123", "Inspector");  
 }  
  
 private async Task SeedSampleInstitutionAndRegistrationAsync()  
 {  
 // Try to find Egypt country to use its Id if exists  
 var egypt = (await \_countryRepository.GetAllAsync(q => q.Where(c => c.TwoLetterIsoCode == "EG"))).FirstOrDefault();  
 var countryId = egypt?.Id ?? 0;  
  
 var inst = (await \_institutionRepository.GetAllAsync(q => q.Where(i => i.LicenseNumber == "LIC-001"))).FirstOrDefault();  
 if (inst == null)  
 {  
 inst = new Institution  
 {  
 Name = "Test Bank Ltd",  
 LicenseNumber = "LIC-001",  
 LicenseTypeId = (int)LicenseType.Banking,  
 LicenseIssueDate = DateTime.UtcNow.AddYears(-1),  
 LicenseExpiryDate = DateTime.UtcNow.AddYears(1),  
 CountryId = countryId,  
 Address = "Cairo, Egypt",  
 IsActive = true,  
 CreatedOnUtc = DateTime.UtcNow  
 };  
 await \_institutionRepository.InsertAsync(inst);  
 }  
  
 var reg = (await \_registrationRepository.GetAllAsync(q => q.Where(r => r.LicenseNumber == "LIC-001"))).FirstOrDefault();  
 if (reg == null)  
 {  
 reg = new Registration  
 {  
 InstitutionName = inst.Name,  
 LicenseNumber = inst.LicenseNumber,  
 LicenseType = inst.LicenseType,  
 LicenseSectorId = (int)LicenseSector.Banking,  
 FinancialDomainId = (int)FinancialDomain.Commercial,  
 IssueDate = inst.LicenseIssueDate,  
 ExpiryDate = inst.LicenseExpiryDate,  
 CountryId = countryId,  
 Address = inst.Address,  
 StatusId = (int)RegistrationStatus.Draft,  
 CreatedByUserId = 0,  
 CreatedOnUtc = DateTime.UtcNow  
 };  
 await \_registrationRepository.InsertAsync(reg);  
 }  
 }  
 }  
}

## InstallUrlMiddleware.cs

using System.Threading.Tasks;  
using Microsoft.AspNetCore.Http;  
  
namespace App.Services.Installation  
{  
 public class InstallUrlMiddleware  
 {  
 private readonly RequestDelegate \_next;  
  
 public InstallUrlMiddleware(RequestDelegate next)  
 {  
 \_next = next;  
 }  
  
 public async Task Invoke(HttpContext context)  
 {  
 // redirect to install URL if needed  
 await \_next(context);  
 }  
 }  
}

## ISO3166.cs

﻿namespace App.Services.Installation;  
  
/// <summary>  
/// Represents the implementation of ISO3166-1  
/// </summary>  
/// <remarks>https://en.wikipedia.org/wiki/List\_of\_ISO\_3166\_country\_codes</remarks>  
public static partial class ISO3166  
{  
 /// <summary>  
 /// Obtain ISO3166-1 Country based on its ISO code.  
 /// </summary>  
 /// <param name="codeISO"></param>  
 /// <returns>ISO3166Country</returns>  
 public static ISO3166Country FromISOCode(int codeISO)  
 {  
 return GetCollection().FirstOrDefault(p => p.NumericCode == codeISO);  
 }  
  
 /// <summary>  
 /// Obtain ISO3166-1 Country based on its alpha-2.  
 /// </summary>  
 /// <param name="countryCode"></param>  
 /// <returns>ISO3166Country</returns>  
 public static ISO3166Country FromCountryCode(string countryCode)  
 {  
 return GetCollection().FirstOrDefault(p => p.Alpha2 == countryCode);  
 }  
  
 /// <summary>  
 /// Collection localization info for country  
 /// </summary>  
 /// <param name="countryCode"></param>  
 /// <returns>IEnumerable<LocalizationInfo></returns>  
 public static IEnumerable<LocalizationInfo> GetLocalizationInfo(string countryCode)  
 {  
 return FromCountryCode(countryCode).LocalizationInfo;  
 }  
  
 #region Collection of counties  
 /// <summary>  
 /// Collection of standard defining codes for the names of countries by ISO 3166-1  
 /// </summary>  
 /// <returns>IEnumerable<ISO3166Country></returns>  
 public static IEnumerable<ISO3166Country> GetCollection()  
 {  
 // This collection built from Wikipedia entry on ISO3166-1 on 8th Dec 2020  
 return new[] {  
 new ISO3166Country("Afghanistan", "AF", "AFG", 4, ["93"]),  
 new ISO3166Country("Åland Islands", "AX", "ALA", 248, ["358"]),  
 new ISO3166Country("Albania", "AL", "ALB", 8, ["355"], localizationInfo: new[] { new LocalizationInfo("sq-AL", "Albanian") }),  
 new ISO3166Country("Algeria", "DZ", "DZA", 12, ["213"], localizationInfo: new[] { new LocalizationInfo("ar-DZ", "Arabic") }),  
 new ISO3166Country("American Samoa", "AS", "ASM", 16, ["1 684"]),  
 new ISO3166Country("Andorra", "AD", "AND", 20, ["376"], localizationInfo: new[] { new LocalizationInfo("ca-ES", "Catalan") }),  
 new ISO3166Country("Angola", "AO", "AGO", 24, ["244"], localizationInfo: new[] { new LocalizationInfo("pt-AO", "Portuguese") }),  
 new ISO3166Country("Anguilla", "AI", "AIA", 660, ["1 264"]),  
 new ISO3166Country("Antarctica", "AQ", "ATA", 10, ["672"]),  
 new ISO3166Country("Antigua and Barbuda", "AG", "ATG", 28, ["1 268"]),  
 new ISO3166Country("Argentina", "AR", "ARG", 32, ["54"], localizationInfo: new[] { new LocalizationInfo("es-AR", "Spanish") }),  
 new ISO3166Country("Armenia", "AM", "ARM", 51, ["374"], localizationInfo: new[] { new LocalizationInfo("hy-AM", "Armenian") }),  
 new ISO3166Country("Aruba", "AW", "ABW", 533, ["297"]),  
 new ISO3166Country("Australia", "AU", "AUS", 36, ["61"]),  
 new ISO3166Country("Austria", "AT", "AUT", 40, ["43"], true, localizationInfo: new[] { new LocalizationInfo("de-AT", "German") }),  
 new ISO3166Country("Azerbaijan", "AZ", "AZE", 31, ["994"], localizationInfo: new[] { new LocalizationInfo("az-Latn-AZ", "Azerbaijani") }),  
 new ISO3166Country("Bahamas", "BS", "BHS", 44, ["1 242"]),  
 new ISO3166Country("Bahrain", "BH", "BHR", 48, ["973"], localizationInfo: new[] { new LocalizationInfo("ar-BH", "Arabic") }),  
 new ISO3166Country("Bangladesh", "BD", "BGD", 50, ["880"], localizationInfo: new[] { new LocalizationInfo("bn-BD", "Bangla") }),  
 new ISO3166Country("Barbados", "BB", "BRB", 52, ["1 246"]),  
 new ISO3166Country("Belarus", "BY", "BLR", 112, ["375"], localizationInfo: new[] { new LocalizationInfo("ru-RU", "Russian") }),  
 new ISO3166Country("Belgium", "BE", "BEL", 56, ["32"], true, localizationInfo: new[] { new LocalizationInfo("fr-BE", "French"), new LocalizationInfo("nl-BE", "Dutch") }),  
 new ISO3166Country("Belize", "BZ", "BLZ", 84, ["501"]),  
 new ISO3166Country("Benin", "BJ", "BEN", 204, ["229"]),  
 new ISO3166Country("Bermuda", "BM", "BMU", 60, ["1 441"]),  
 new ISO3166Country("Bhutan", "BT", "BTN", 64, ["975"]),  
 new ISO3166Country("Bolivia (Plurinational State of)", "BO", "BOL", 68, ["591"], localizationInfo: new[] { new LocalizationInfo("es-BO", "Spanish") }),  
 new ISO3166Country("Bonaire, Sint Eustatius and Saba", "BQ", "BES", 535, ["599"]),  
 new ISO3166Country("Bosnia and Herzegovina", "BA", "BIH", 70, ["387"]),  
 new ISO3166Country("Botswana", "BW", "BWA", 72, ["267"]),  
 new ISO3166Country("Bouvet Island", "BV", "BVT", 74),  
 new ISO3166Country("Brazil", "BR", "BRA", 76, ["55"], localizationInfo: new[] { new LocalizationInfo("pt-BR", "Portuguese") }),  
 new ISO3166Country("British Indian Ocean Territory", "IO", "IOT", 86, ["246"]),  
 new ISO3166Country("Brunei Darussalam", "BN", "BRN", 96, ["673"]),  
 new ISO3166Country("Bulgaria", "BG", "BGR", 100, ["359"], true, localizationInfo: new[] { new LocalizationInfo("bg-BG", "Bulgarian") }),  
 new ISO3166Country("Burkina Faso", "BF", "BFA", 854, ["226"]),  
 new ISO3166Country("Burundi", "BI", "BDI", 108, ["257"]),  
 new ISO3166Country("Cabo Verde", "CV", "CPV", 132, ["238"], localizationInfo: new[] { new LocalizationInfo("pt-CV", "Portuguese") }),  
 new ISO3166Country("Cambodia", "KH", "KHM", 116, ["855"]),  
 new ISO3166Country("Cameroon", "CM", "CMR", 120, ["237"]),  
 new ISO3166Country("Canada", "CA", "CAN", 124, ["1"], localizationInfo: new[] { new LocalizationInfo("en-US", "English"), new LocalizationInfo("fr-FR", "French") }),  
 new ISO3166Country("Cayman Islands", "KY", "CYM", 136, ["1 345"]),  
 new ISO3166Country("Central African Republic", "CF", "CAF", 140, ["236"]),  
 new ISO3166Country("Chad", "TD", "TCD", 148, ["235"]),  
 new ISO3166Country("Chile", "CL", "CHL", 152, ["56"], localizationInfo: new[] { new LocalizationInfo("es-CL", "Spanish") }),  
 new ISO3166Country("China", "CN", "CHN", 156, ["86"], localizationInfo: new[] { new LocalizationInfo("zh-CN", "Chinese") }),  
 new ISO3166Country("Christmas Island", "CX", "CXR", 162, ["61"]),  
 new ISO3166Country("Cocos (Keeling) Islands", "CC", "CCK", 166, ["61"]),  
 new ISO3166Country("Colombia", "CO", "COL", 170, ["57"], localizationInfo: new[] { new LocalizationInfo("es-CO", "Spanish") }),  
 new ISO3166Country("Comoros", "KM", "COM", 174, ["269"]),  
 new ISO3166Country("Congo", "CG", "COG", 178, ["242"]),  
 new ISO3166Country("Congo (Democratic Republic of the)", "CD", "COD", 180, ["243"]),  
 new ISO3166Country("Cook Islands", "CK", "COK", 184, ["682"]),  
 new ISO3166Country("Costa Rica", "CR", "CRI", 188, ["506"], localizationInfo: new[] { new LocalizationInfo("es-CR", "Spanish") }),  
 new ISO3166Country("Côte d'Ivoire", "CI", "CIV", 384, ["225"]),  
 new ISO3166Country("Croatia", "HR", "HRV", 191, ["385"], true, localizationInfo: new[] { new LocalizationInfo("hr-HR", "Croatian") }),  
 new ISO3166Country("Cuba", "CU", "CUB", 192, ["53"], localizationInfo: new[] { new LocalizationInfo("es-CU", "Spanish") }),  
 new ISO3166Country("Curaçao", "CW", "CUW", 531, ["599"]),  
 new ISO3166Country("Cyprus", "CY", "CYP", 196, ["357"], true, localizationInfo: new[] { new LocalizationInfo("el-CY", "Greek"), new LocalizationInfo("tr-CY", "Turkish") }),  
 new ISO3166Country("Czechia", "CZ", "CZE", 203, ["420"], true, localizationInfo: new[] { new LocalizationInfo("cs-CZ", "Czech") }),  
 new ISO3166Country("Denmark", "DK", "DNK", 208, ["45"], true, localizationInfo: new[] { new LocalizationInfo("da-DK", "Danish") }),  
 new ISO3166Country("Djibouti", "DJ", "DJI", 262, ["253"]),  
 new ISO3166Country("Dominica", "DM", "DMA", 212, ["1 767"]),  
 new ISO3166Country("Dominican Republic", "DO", "DOM", 214, ["1 809", "1 829", "1 849"], localizationInfo: new[] { new LocalizationInfo("es-DO", "Spanish") }),  
 new ISO3166Country("Ecuador", "EC", "ECU", 218, ["593"], localizationInfo: new[] { new LocalizationInfo("es-EC", "Spanish") }),  
 new ISO3166Country("Egypt", "EG", "EGY", 818, ["20"], localizationInfo: new[] { new LocalizationInfo("ar-EG", "Arabic") }),  
 new ISO3166Country("El Salvador", "SV", "SLV", 222, ["503"], localizationInfo: new[] { new LocalizationInfo("es-SV", "Spanish") }),  
 new ISO3166Country("Equatorial Guinea", "GQ", "GNQ", 226, ["240"]),  
 new ISO3166Country("Eritrea", "ER", "ERI", 232, ["291"]),  
 new ISO3166Country("Estonia", "EE", "EST", 233, ["372"], true, localizationInfo: new[] { new LocalizationInfo("et-EE", "Estonian") }),  
 new ISO3166Country("Eswatini", "SZ", "SWZ", 748, ["268"]),  
 new ISO3166Country("Ethiopia", "ET", "ETH", 231, ["251"]),  
 new ISO3166Country("Falkland Islands (Malvinas)", "FK", "FLK", 238, ["500"]),  
 new ISO3166Country("Faroe Islands", "FO", "FRO", 234, ["298"]),  
 new ISO3166Country("Fiji", "FJ", "FJI", 242, ["679"]),  
 new ISO3166Country("Finland", "FI", "FIN", 246, ["358"], true, localizationInfo: new[] { new LocalizationInfo("fi-FI", "Finnish") }),  
 new ISO3166Country("France", "FR", "FRA", 250, ["33"], true, localizationInfo: new[] { new LocalizationInfo("fr-FR", "French") }),  
 new ISO3166Country("French Guiana", "GF", "GUF", 254, ["594"]),  
 new ISO3166Country("French Polynesia", "PF", "PYF", 258, ["689"]),  
 new ISO3166Country("French Southern Territories", "TF", "ATF", 260, ["262"]),  
 new ISO3166Country("Gabon", "GA", "GAB", 266, ["241"]),  
 new ISO3166Country("Gambia", "GM", "GMB", 270, ["220"]),  
 new ISO3166Country("Georgia", "GE", "GEO", 268, ["995"], localizationInfo: new[] { new LocalizationInfo("ka-GE", "Georgian") }),  
 new ISO3166Country("Germany", "DE", "DEU", 276, ["49"], true, localizationInfo: new[] { new LocalizationInfo("de-DE", "German") }),  
 new ISO3166Country("Ghana", "GH", "GHA", 288, ["233"]),  
 new ISO3166Country("Gibraltar", "GI", "GIB", 292, ["350"]),  
 new ISO3166Country("Greece", "GR", "GRC", 300, ["30"], true, localizationInfo: new[] { new LocalizationInfo("el-GR", "Greek") }),  
 new ISO3166Country("Greenland", "GL", "GRL", 304, ["299"]),  
 new ISO3166Country("Grenada", "GD", "GRD", 308, ["1 473"]),  
 new ISO3166Country("Guadeloupe", "GP", "GLP", 312, ["590"]),  
 new ISO3166Country("Guam", "GU", "GUM", 316, ["1 671"]),  
 new ISO3166Country("Guatemala", "GT", "GTM", 320, ["502"], localizationInfo: new[] { new LocalizationInfo("es-GT", "Spanish") }),  
 new ISO3166Country("Guernsey", "GG", "GGY", 831, ["44 1481"]),  
 new ISO3166Country("Guinea", "GN", "GIN", 324, ["224"]),  
 new ISO3166Country("Guinea-Bissau", "GW", "GNB", 624, ["245"], localizationInfo: new[] { new LocalizationInfo("pt-GW", "Portuguese") }),  
 new ISO3166Country("Guyana", "GY", "GUY", 328, ["592"]),  
 new ISO3166Country("Haiti", "HT", "HTI", 332, ["509"]),  
 new ISO3166Country("Heard Island and McDonald Islands", "HM", "HMD", 334),  
 new ISO3166Country("Holy See", "VA", "VAT", 336, ["379"]),  
 new ISO3166Country("Honduras", "HN", "HND", 340, ["504"], localizationInfo: new[] { new LocalizationInfo("es-HN", "Spanish") }),  
 new ISO3166Country("Hong Kong", "HK", "HKG", 344, ["852"], localizationInfo: new[] { new LocalizationInfo("zh-CN", "Chinese") }),  
 new ISO3166Country("Hungary", "HU", "HUN", 348, ["36"], true, localizationInfo: new[] { new LocalizationInfo("hu-HU", "Hungarian") }),  
 new ISO3166Country("Iceland", "IS", "ISL", 352, ["354"], localizationInfo: new[] { new LocalizationInfo("is-IS", "Icelandic") }),  
 new ISO3166Country("India", "IN", "IND", 356, ["91"]),  
 new ISO3166Country("Indonesia", "ID", "IDN", 360, ["62"], localizationInfo: new[] { new LocalizationInfo("id-ID", "Indonesian") }),  
 new ISO3166Country("Iran (Islamic Republic of)", "IR", "IRN", 364, ["98"], localizationInfo: new[] { new LocalizationInfo("fa-IR", "Persian") }),  
 new ISO3166Country("Iraq", "IQ", "IRQ", 368, ["964"], localizationInfo: new[] { new LocalizationInfo("ar-IQ", "Arabic") }),  
 new ISO3166Country("Ireland", "IE", "IRL", 372, ["353"], true),  
 new ISO3166Country("Isle of Man", "IM", "IMN", 833, ["44 1624"]),  
 new ISO3166Country("Israel", "IL", "ISR", 376, ["972"], localizationInfo: new[] { new LocalizationInfo("he-IL", "Hebrew") }),  
 new ISO3166Country("Italy", "IT", "ITA", 380, ["39"], true, localizationInfo: new[] { new LocalizationInfo("it-IT", "Italian") }),  
 new ISO3166Country("Jamaica", "JM", "JAM", 388, ["1 876"]),  
 new ISO3166Country("Japan", "JP", "JPN", 392, ["81"], localizationInfo: new[] { new LocalizationInfo("ja-JP", "Japanese") }),  
 new ISO3166Country("Jersey", "JE", "JEY", 832, ["44 1534"]),  
 new ISO3166Country("Jordan", "JO", "JOR", 400, ["962"], localizationInfo: new[] { new LocalizationInfo("ar-JO", "Arabic") }),  
 new ISO3166Country("Kazakhstan", "KZ", "KAZ", 398, ["7"]),  
 new ISO3166Country("Kenya", "KE", "KEN", 404, ["254"]),  
 new ISO3166Country("Kiribati", "KI", "KIR", 296, ["686"]),  
 new ISO3166Country("Korea (Democratic People's Republic of)", "KP", "PRK", 408, ["850"]),  
 new ISO3166Country("Korea (Republic of)", "KR", "KOR", 410, ["82"]),  
 new ISO3166Country("Kuwait", "KW", "KWT", 414, ["965"], localizationInfo: new[] { new LocalizationInfo("ar-KW", "Arabic") }),  
 new ISO3166Country("Kyrgyzstan", "KG", "KGZ", 417, ["996"], localizationInfo: new[] { new LocalizationInfo("ky-KG", "Kyrgyz") }),  
 new ISO3166Country("Lao People's Democratic Republic", "LA", "LAO", 418, ["856"]),  
 new ISO3166Country("Latvia", "LV", "LVA", 428, ["371"], true, localizationInfo: new[] { new LocalizationInfo("lv-LV", "Latvian") }),  
 new ISO3166Country("Lebanon", "LB", "LBN", 422, ["961"], localizationInfo: new[] { new LocalizationInfo("ar-LB", "Arabic") }),  
 new ISO3166Country("Lesotho", "LS", "LSO", 426, ["266"]),  
 new ISO3166Country("Liberia", "LR", "LBR", 430, ["231"]),  
 new ISO3166Country("Libya", "LY", "LBY", 434, ["218"], localizationInfo: new[] { new LocalizationInfo("ar-LY", "Arabic") }),  
 new ISO3166Country("Liechtenstein", "LI", "LIE", 438, ["423"], localizationInfo: new[] { new LocalizationInfo("de-LI", "German") }),  
 new ISO3166Country("Lithuania", "LT", "LTU", 440, ["370"], true, localizationInfo: new[] { new LocalizationInfo("lt-LT", "Lithuanian") }),  
 new ISO3166Country("Luxembourg", "LU", "LUX", 442, ["352"], true, localizationInfo: new[] { new LocalizationInfo("fr-FR", "French"), new LocalizationInfo("de-LU", "German") }),  
 new ISO3166Country("Macao", "MO", "MAC", 446, ["853"], localizationInfo: new[] { new LocalizationInfo("zh-CN", "Chinese") }),  
 new ISO3166Country("North Macedonia", "MK", "MKD", 807, ["389"], localizationInfo: new[] { new LocalizationInfo("mk-MK", "Macedonian") }),  
 new ISO3166Country("Madagascar", "MG", "MDG", 450, ["261"]),  
 new ISO3166Country("Malawi", "MW", "MWI", 454, ["265"]),  
 new ISO3166Country("Malaysia", "MY", "MYS", 458, ["60"]),  
 new ISO3166Country("Maldives", "MV", "MDV", 462, ["960"]),  
 new ISO3166Country("Mali", "ML", "MLI", 466, ["223"]),  
 new ISO3166Country("Malta", "MT", "MLT", 470, ["356"], true),  
 new ISO3166Country("Marshall Islands", "MH", "MHL", 584, ["692"]),  
 new ISO3166Country("Martinique", "MQ", "MTQ", 474, ["596"]),  
 new ISO3166Country("Mauritania", "MR", "MRT", 478, ["222"]),  
 new ISO3166Country("Mauritius", "MU", "MUS", 480, ["230"]),  
 new ISO3166Country("Mayotte", "YT", "MYT", 175, ["262"]),  
 new ISO3166Country("Mexico", "MX", "MEX", 484, ["52"], localizationInfo: new[] { new LocalizationInfo("es-MX", "Spanish") }),  
 new ISO3166Country("Micronesia (Federated States of)", "FM", "FSM", 583, ["691"]),  
 new ISO3166Country("Moldova (Republic of)", "MD", "MDA", 498, ["373"]),  
 new ISO3166Country("Monaco", "MC", "MCO", 492, ["377"], localizationInfo: new[] { new LocalizationInfo("fr-FR", "French") }),  
 new ISO3166Country("Mongolia", "MN", "MNG", 496, ["976"]),  
 new ISO3166Country("Montenegro", "ME", "MNE", 499, ["382"]),  
 new ISO3166Country("Montserrat", "MS", "MSR", 500, ["1 664"]),  
 new ISO3166Country("Morocco", "MA", "MAR", 504, ["212"], localizationInfo: new[] { new LocalizationInfo("ar-MA", "Arabic") }),  
 new ISO3166Country("Mozambique", "MZ", "MOZ", 508, ["258"], localizationInfo: new[] { new LocalizationInfo("pt-MZ", "Portuguese") }),  
 new ISO3166Country("Myanmar", "MM", "MMR", 104, ["95"]),  
 new ISO3166Country("Namibia", "NA", "NAM", 516, ["264"]),  
 new ISO3166Country("Nauru", "NR", "NRU", 520, ["674"]),  
 new ISO3166Country("Nepal", "NP", "NPL", 524, ["977"], localizationInfo: new[] { new LocalizationInfo("ne-NP", "Nepali") }),  
 new ISO3166Country("Netherlands", "NL", "NLD", 528, ["31"], true, localizationInfo: new[] { new LocalizationInfo("nl-NL", "Dutch") }),  
 new ISO3166Country("New Caledonia", "NC", "NCL", 540, ["687"]),  
 new ISO3166Country("New Zealand", "NZ", "NZL", 554, ["64"]),  
 new ISO3166Country("Nicaragua", "NI", "NIC", 558, ["505"], localizationInfo: new[] { new LocalizationInfo("es-NI", "Spanish") }),  
 new ISO3166Country("Niger", "NE", "NER", 562, ["227"]),  
 new ISO3166Country("Nigeria", "NG", "NGA", 566, ["234"]),  
 new ISO3166Country("Niue", "NU", "NIU", 570, ["683"]),  
 new ISO3166Country("Norfolk Island", "NF", "NFK", 574, ["672"]),  
 new ISO3166Country("Northern Mariana Islands", "MP", "MNP", 580, ["1 670"]),  
 new ISO3166Country("Norway", "NO", "NOR", 578, ["47"], localizationInfo: new[] { new LocalizationInfo("nb-NO", "Norwegian") }),  
 new ISO3166Country("Oman", "OM", "OMN", 512, ["968"], localizationInfo: new[] { new LocalizationInfo("ar-OM", "Arabic") }),  
 new ISO3166Country("Pakistan", "PK", "PAK", 586, ["92"]),  
 new ISO3166Country("Palau", "PW", "PLW", 585, ["680"]),  
 new ISO3166Country("Palestine, State of", "PS", "PSE", 275, ["970"], localizationInfo: new[] { new LocalizationInfo("ar-PS", "Arabic") }),  
 new ISO3166Country("Panama", "PA", "PAN", 591, ["507"], localizationInfo: new[] { new LocalizationInfo("es-PA", "Spanish") }),  
 new ISO3166Country("Papua New Guinea", "PG", "PNG", 598, ["675"]),  
 new ISO3166Country("Paraguay", "PY", "PRY", 600, ["595"], localizationInfo: new[] { new LocalizationInfo("es-PY", "Spanish") }),  
 new ISO3166Country("Peru", "PE", "PER", 604, ["51"], localizationInfo: new[] { new LocalizationInfo("es-PE", "Spanish") }),  
 new ISO3166Country("Philippines", "PH", "PHL", 608, ["63"]),  
 new ISO3166Country("Pitcairn", "PN", "PCN", 612, ["64"]),  
 new ISO3166Country("Poland", "PL", "POL", 616, ["48"], true, localizationInfo: new[] { new LocalizationInfo("pl-PL", "Polish") }),  
 new ISO3166Country("Portugal", "PT", "PRT", 620, ["351"], true),  
 new ISO3166Country("Puerto Rico", "PR", "PRI", 630, ["1 787", "1 939"], localizationInfo: new[] { new LocalizationInfo("es-PR", "Spanish") }),  
 new ISO3166Country("Qatar", "QA", "QAT", 634, ["974"], localizationInfo: new[] { new LocalizationInfo("ar-QA", "Arabic") }),  
 new ISO3166Country("Réunion", "RE", "REU", 638, ["262"]),  
 new ISO3166Country("Romania", "RO", "ROU", 642, ["40"], true, localizationInfo: new[] { new LocalizationInfo("ro-RO", "Romanian") }),  
 new ISO3166Country("Russian Federation", "RU", "RUS", 643, ["7"], localizationInfo: new[] { new LocalizationInfo("ru-RU", "Russian") }),  
 new ISO3166Country("Rwanda", "RW", "RWA", 646, ["250"]),  
 new ISO3166Country("Saint Barthélemy", "BL", "BLM", 652, ["590"]),  
 new ISO3166Country("Saint Helena, Ascension and Tristan da Cunha", "SH", "SHN", 654, ["290"]),  
 new ISO3166Country("Saint Kitts and Nevis", "KN", "KNA", 659, ["1 869"]),  
 new ISO3166Country("Saint Lucia", "LC", "LCA", 662, ["1 758"]),  
 new ISO3166Country("Saint Martin (French part)", "MF", "MAF", 663, ["590"]),  
 new ISO3166Country("Saint Pierre and Miquelon", "PM", "SPM", 666, ["508"]),  
 new ISO3166Country("Saint Vincent and the Grenadines", "VC", "VCT", 670, ["1 784"]),  
 new ISO3166Country("Samoa", "WS", "WSM", 882, ["685"]),  
 new ISO3166Country("San Marino", "SP", "SMR", 674),  
 new ISO3166Country("Sao Tome and Principe", "ST", "STP", 678, ["239"]),  
 new ISO3166Country("Saudi Arabia", "SA", "SAU", 682, ["966"], localizationInfo: new[] { new LocalizationInfo("ar-SA", "Arabic") }),  
 new ISO3166Country("Senegal", "SN", "SEN", 686, ["221"]),  
 new ISO3166Country("Serbia", "RS", "SRB", 688, ["381"], localizationInfo: new[] { new LocalizationInfo("sr-Cyrl-RS", "Serbian (Cyrillic)"), new LocalizationInfo("sr-Latn-RS", "Serbian (Latin)") }),  
 new ISO3166Country("Seychelles", "SC", "SYC", 690, ["248"]),  
 new ISO3166Country("Sierra Leone", "SL", "SLE", 694, ["232"]),  
 new ISO3166Country("Singapore", "SG", "SGP", 702, ["65"]),  
 new ISO3166Country("Sint Maarten (Dutch part)", "SX", "SXM", 534, ["1 721"]),  
 new ISO3166Country("Slovakia", "SK", "SVK", 703, ["421"], true, localizationInfo: new[] { new LocalizationInfo("sk-SK", "Slovak") }),  
 new ISO3166Country("Slovenia", "SI", "SVN", 705, ["386"], true, localizationInfo: new[] { new LocalizationInfo("sl-SI", "Slovenian") }),  
 new ISO3166Country("Solomon Islands", "SB", "SLB", 90, ["677"]),  
 new ISO3166Country("Somalia", "SO", "SOM", 706, ["252"]),  
 new ISO3166Country("South Africa", "ZA", "ZAF", 710, ["27"]),  
 new ISO3166Country("South Georgia and the South Sandwich Islands", "GS", "SGS", 239, ["500"]),  
 new ISO3166Country("South Sudan", "SS", "SSD", 728, ["211"]),  
 new ISO3166Country("Spain", "ES", "ESP", 724, ["34"], true, localizationInfo: new[] { new LocalizationInfo("ca-ES", "Valencian"), new LocalizationInfo("es-ES", "Spanish") }),  
 new ISO3166Country("Sri Lanka", "LK", "LKA", 144, ["94"], localizationInfo: new[] { new LocalizationInfo("si-LK", "Sinhala") }),  
 new ISO3166Country("Sudan", "SD", "SDN", 729, ["249"]),  
 new ISO3166Country("Suriname", "SR", "SUR", 740, ["597"]),  
 new ISO3166Country("Svalbard and Jan Mayen", "SJ", "SJM", 744, ["47"]),  
 new ISO3166Country("Sweden", "SE", "SWE", 752, ["46"], true, localizationInfo: new[] { new LocalizationInfo("sv-SE", "Swedish") }),  
 new ISO3166Country("Switzerland", "CH", "CHE", 756, ["41"], localizationInfo: new[] { new LocalizationInfo("de-CH", "German"), new LocalizationInfo("fr-CH", "French") }),  
 new ISO3166Country("Syrian Arab Republic", "SY", "SYR", 760, ["963"], localizationInfo: new[] { new LocalizationInfo("ar-SY", "Arabic") }),  
 new ISO3166Country("Taiwan, Province of China", "TW", "TWN", 158, ["886"]),  
 new ISO3166Country("Tajikistan", "TJ", "TJK", 762, ["992"]),  
 new ISO3166Country("Tanzania, United Republic of", "TZ", "TZA", 834, ["255"]),  
 new ISO3166Country("Thailand", "TH", "THA", 764, ["66"], localizationInfo: new[] { new LocalizationInfo("th-TH", "Thai") }),  
 new ISO3166Country("Timor-Leste", "TL", "TLS", 626, ["670"]),  
 new ISO3166Country("Togo", "TG", "TGO", 768, ["228"]),  
 new ISO3166Country("Tokelau", "TK", "TKL", 772, ["690"]),  
 new ISO3166Country("Tonga", "TO", "TON", 776, ["676"]),  
 new ISO3166Country("Trinidad and Tobago", "TT", "TTO", 780, ["1 868"]),  
 new ISO3166Country("Tunisia", "TN", "TUN", 788, ["216"], localizationInfo: new[] { new LocalizationInfo("ar-TN", "Arabic") }),  
 new ISO3166Country("Turkey", "TR", "TUR", 792, ["90"], localizationInfo: new[] { new LocalizationInfo("tr-TR", "Turkish") }),  
 new ISO3166Country("Turkmenistan", "TM", "TKM", 795, ["993"]),  
 new ISO3166Country("Turks and Caicos Islands", "TC", "TCA", 796, ["1 649"]),  
 new ISO3166Country("Tuvalu", "TV", "TUV", 798, ["688"]),  
 new ISO3166Country("Uganda", "UG", "UGA", 800, ["256"]),  
 new ISO3166Country("Ukraine", "UA", "UKR", 804, ["380"], localizationInfo: new[] { new LocalizationInfo("uk-UA", "Ukrainian"), new LocalizationInfo("ru-RU", "Russian") }),  
 new ISO3166Country("United Arab Emirates", "AE", "ARE", 784, ["971"], localizationInfo: new[] { new LocalizationInfo("ar-AE", "Arabic") }),  
 new ISO3166Country("United Kingdom of Great Britain and Northern Ireland", "GB", "GBR", 826, ["44"]),  
 new ISO3166Country("United States Minor Outlying Islands", "UM", "UMI", 581),  
 new ISO3166Country("United States of America", "US", "USA", 840, ["1"]),  
 new ISO3166Country("Uruguay", "UY", "URY", 858, ["598"], localizationInfo: new[] { new LocalizationInfo("es-UY", "Spanish") }),  
 new ISO3166Country("Uzbekistan", "UZ", "UZB", 860, ["998"]),  
 new ISO3166Country("Vanuatu", "VU", "VUT", 548, ["678"]),  
 new ISO3166Country("Venezuela (Bolivarian Republic of)", "VE", "VEN", 862, ["58"], localizationInfo: new[] { new LocalizationInfo("es-VE", "Spanish") }),  
 new ISO3166Country("Vietnam", "VN", "VNM", 704, ["84"], localizationInfo: new[] { new LocalizationInfo("vi-VN", "Vietnamese") }),  
 new ISO3166Country("Virgin Islands (British)", "VG", "VGB", 92, ["1 284"]),  
 new ISO3166Country("Virgin Islands (U.S.)", "VI", "VIR", 850, ["1 340"]),  
 new ISO3166Country("Wallis and Futuna", "WF", "WLF", 876, ["681"]),  
 new ISO3166Country("Western Sahara", "EH", "ESH", 732, ["212"]),  
 new ISO3166Country("Yemen", "YE", "YEM", 887, ["967"], localizationInfo: new[] { new LocalizationInfo("ar-YE", "Arabic") }),  
 new ISO3166Country("Zambia", "ZM", "ZMB", 894, ["260"]),  
 new ISO3166Country("Zimbabwe", "ZW", "ZWE", 716, ["263"])  
 };  
 }  
 #endregion  
}  
  
/// <summary>  
/// Representation of an ISO3166-1 Country  
/// </summary>  
public partial class ISO3166Country  
{  
 public ISO3166Country(string name, string alpha2, string alpha3, int numericCode, string[] dialCodes = null, bool subjectToVat = false, IEnumerable<LocalizationInfo> localizationInfo = null)  
 {  
 Name = name;  
 Alpha2 = alpha2;  
 Alpha3 = alpha3;  
 NumericCode = numericCode;  
 DialCodes = dialCodes;  
 SubjectToVat = subjectToVat;  
 LocalizationInfo = localizationInfo ?? (new[] { new LocalizationInfo("en-US", "English") });  
 }  
  
 /// <summary>  
 ///English short name of country  
 /// </summary>  
 public string Name { get; protected set; }  
  
 /// <summary>  
 /// Two-letter country code  
 /// </summary>  
 public string Alpha2 { get; protected set; }  
  
 /// <summary>  
 /// three-letter country code which allow a better visual association between the codes and the country names than the alpha-2 codes  
 /// </summary>  
 public string Alpha3 { get; protected set; }  
  
 /// <summary>  
 /// Three-digit country code which are identical to those developed and maintained by the United Nations Statistics Division  
 /// </summary>  
 public int NumericCode { get; protected set; }  
  
 /// <summary>  
 /// Phone codes  
 /// </summary>  
 public string[] DialCodes { get; protected set; }  
  
 /// <summary>  
 /// Belonging to the European Union  
 /// </summary>  
 public bool SubjectToVat { get; protected set; }  
  
 public IEnumerable<LocalizationInfo> LocalizationInfo { get; protected set; }  
}  
  
public partial class LocalizationInfo  
{  
 public LocalizationInfo(string culture, string language)  
 {  
 Culture = culture;  
 Language = language;  
 }  
  
 public string Culture { get; protected set; }  
  
 public string Language { get; protected set; }  
}

## IInstitutionService.cs

using App.Core.Domain.Institutions;  
using App.Core.Domain.Ref;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
using System;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace App.Services.Institutions  
{  
 public interface IInstitutionService  
 {  
 // CRUD  
 Task<Institution> GetByIdAsync(int id);  
 Task<IList<Institution>> GetAllAsync();  
 Task<ServiceResult<Institution>> InsertAsync(Institution institution);  
 Task<ServiceResult<Institution>> UpdateAsync(Institution institution);  
 Task<ServiceResult> DeleteAsync(int id);  
  
 // Checks  
 Task<bool> ExistsAsync(string institutionName, string licenseNumber);  
 Task<bool> IsValidForRegistrationAsync(int institutionId);  
  
 // Relations  
 Task<IList<Registration>> GetRegistrationsAsync(int institutionId);  
 Task<Registration> GetLatestRegistrationAsync(int institutionId);  
  
 // Search  
 Task<PagedResult<Institution>> SearchAsync(  
 string name = null,  
 string licenseNumber = null,  
 int? countryId = null,  
 LicenseSector? licenseSector = null,  
 FinancialDomain? financialDomain = null,  
 int pageIndex = 0,  
 int pageSize = int.MaxValue  
 );  
  
 // Reporting  
 Task<IDictionary<LicenseSector, int>> GetInstitutionCountBySectorAsync();  
 Task<IDictionary<int, int>> GetInstitutionCountByCountryAsync();  
 }  
}

## InstitutionService.cs

using App.Core.Domain.Institutions;  
using App.Core.Domain.Ref;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Institutions  
{  
 public class InstitutionService : BaseService, IInstitutionService  
 {  
 private readonly IRepository<Institution> \_institutionRepository;  
 private readonly IRepository<Registration> \_registrationRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public InstitutionService(  
 IRepository<Institution> institutionRepository,  
 IRepository<Registration> registrationRepository,  
 ILocalizationService localizationService)  
 {  
 \_institutionRepository = institutionRepository;  
 \_registrationRepository = registrationRepository;  
 \_localizationService = localizationService;  
 }  
  
 #region CRUD  
  
 public async Task<Institution> GetByIdAsync(int id)  
 => await \_institutionRepository.GetByIdAsync(id);  
  
 public async Task<IList<Institution>> GetAllAsync()  
 => await \_institutionRepository.GetAllAsync(q => q.OrderBy(i => i.Name));  
  
 public async Task<ServiceResult<Institution>> InsertAsync(Institution institution)  
 {  
 if (institution == null)  
 return Failed<Institution>(await \_localizationService.GetResourceAsync("Institution.Insert.Null"));  
  
 if (await ExistsAsync(institution.Name, institution.LicenseNumber))  
 return Failed<Institution>(await \_localizationService.GetResourceAsync("Institution.Insert.Duplicate"));  
  
 institution.CreatedOnUtc = DateTime.UtcNow;  
 await \_institutionRepository.InsertAsync(institution);  
  
 return Success(institution, await \_localizationService.GetResourceAsync("Institution.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<Institution>> UpdateAsync(Institution institution)  
 {  
 if (institution == null)  
 return Failed<Institution>(await \_localizationService.GetResourceAsync("Institution.Update.Null"));  
  
 institution.UpdatedOnUtc = DateTime.UtcNow;  
 await \_institutionRepository.UpdateAsync(institution);  
  
 return Success(institution, await \_localizationService.GetResourceAsync("Institution.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var institution = await \_institutionRepository.GetByIdAsync(id);  
 if (institution == null)  
 return Failed(await \_localizationService.GetResourceAsync("Institution.NotFound"));  
  
 await \_institutionRepository.DeleteAsync(institution);  
  
 return Success(await \_localizationService.GetResourceAsync("Institution.Delete.Success"));  
 }  
  
 #endregion  
  
 #region Checks  
  
 public async Task<bool> ExistsAsync(string institutionName, string licenseNumber)  
 {  
 var institutions = await \_institutionRepository.GetAllAsync(q =>  
 q.Where(i => i.Name == institutionName || i.LicenseNumber == licenseNumber));  
 return institutions != null && institutions.Count > 0;  
 }  
  
 public async Task<bool> IsValidForRegistrationAsync(int institutionId)  
 {  
 var institution = await \_institutionRepository.GetByIdAsync(institutionId);  
 if (institution == null) return false;  
  
 // Example rule: must not be archived & license valid  
 return institution.IsActive && institution.LicenseExpiryDate > DateTime.UtcNow;  
 }  
  
 #endregion  
  
 #region Relations  
  
 public async Task<IList<Registration>> GetRegistrationsAsync(int institutionId)  
 => await \_registrationRepository.GetAllAsync(q => q.Where(r => r.InstitutionId == institutionId));  
  
 public async Task<Registration> GetLatestRegistrationAsync(int institutionId)  
 {  
 var registrations = await \_registrationRepository.GetAllAsync(q =>  
 q.Where(r => r.InstitutionId == institutionId)  
 .OrderByDescending(r => r.CreatedOnUtc));  
 return registrations.FirstOrDefault();  
 }  
  
 #endregion  
  
 #region Search  
  
 public async Task<PagedResult<Institution>> SearchAsync(  
 string name = null,  
 string licenseNumber = null,  
 int? countryId = null,  
 LicenseSector? licenseSector = null,  
 FinancialDomain? financialDomain = null,  
 int pageIndex = 0,  
 int pageSize = int.MaxValue)  
 {  
 var query = \_institutionRepository.Table;  
  
 if (!string.IsNullOrEmpty(name))  
 query = query.Where(i => i.Name.Contains(name));  
  
 if (!string.IsNullOrEmpty(licenseNumber))  
 query = query.Where(i => i.LicenseNumber.Contains(licenseNumber));  
  
 if (countryId.HasValue)  
 query = query.Where(i => i.CountryId == countryId);  
  
 if (licenseSector.HasValue)  
 query = query.Where(i => i.LicenseSector == licenseSector);  
  
 if (financialDomain.HasValue)  
 query = query.Where(i => i.FinancialDomain == financialDomain);  
  
 var totalCount = query.Count();  
  
 var items = query  
 .OrderBy(i => i.Name)  
 .Skip(pageIndex \* pageSize)  
 .Take(pageSize)  
 .ToList();  
  
 return await Task.FromResult(new PagedResult<Institution>(items, totalCount, pageIndex, pageSize));  
 }  
  
 public async Task<IDictionary<LicenseSector, int>> GetInstitutionCountBySectorAsync()  
 {  
 var institutions = await \_institutionRepository.GetAllAsync(q => q, true);  
 return institutions  
 .GroupBy(i => i.LicenseSector)  
 .ToDictionary(g => g.Key, g => g.Count());  
 }  
  
 public async Task<IDictionary<int, int>> GetInstitutionCountByCountryAsync()  
 {  
 var institutions = await \_institutionRepository.GetAllAsync(q => q, true);  
 return institutions  
 .GroupBy(i => i.CountryId)  
 .ToDictionary(g => g.Key, g => g.Count());  
 }  
  
 #endregion  
 }  
}

## ILookupService.cs

﻿using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Ref;  
  
namespace App.Services.Lookup  
{  
 public interface ILookupService  
 {  
 // Business Scale Ranges  
 Task<IList<BusinessScaleRange>> GetBusinessScaleRangesAsync();  
 Task<BusinessScaleRange> GetBusinessScaleRangeByIdAsync(int id);  
 Task<BusinessScaleRange> FindBusinessScaleRangeAsync(int employeesCount);  
  
 // Employee Ranges  
 Task<IList<EmployeeRange>> GetEmployeeRangesAsync();  
 Task<EmployeeRange> GetEmployeeRangeByIdAsync(int id);  
 Task<EmployeeRange> FindEmployeeRangeAsync(int employeesCount);  
  
 // Enums  
 Task<IList<string>> GetLicenseTypesAsync();  
 Task<IList<string>> GetLicenseSectorsAsync();  
 Task<IList<string>> GetFinancialDomainsAsync();  
  
 // Invalidate cache (for Admin actions)  
 Task ClearLookupCacheAsync();  
 }  
}

## LookupService.cs

﻿using App.Core.Caching;  
using App.Core.Domain.Ref;  
using App.Core.RepositoryServices;  
using App.Data;  
using Microsoft.EntityFrameworkCore;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Lookup  
{  
 public class LookupService : ILookupService  
 {  
 private readonly IRepository<BusinessScaleRange> \_businessScaleRepo;  
 private readonly IRepository<EmployeeRange> \_employeeRangeRepo;  
 private readonly IStaticCacheManager \_cache;  
  
 // CacheKey definitions  
 private static readonly CacheKey BusinessScaleCacheKey = new("Lookup.BusinessScaleRanges");  
 private static readonly CacheKey EmployeeRangeCacheKey = new("Lookup.EmployeeRanges");  
  
 public LookupService(  
 IRepository<BusinessScaleRange> businessScaleRepo,  
 IRepository<EmployeeRange> employeeRangeRepo,  
 IStaticCacheManager cache)  
 {  
 \_businessScaleRepo = businessScaleRepo;  
 \_employeeRangeRepo = employeeRangeRepo;  
 \_cache = cache;  
 }  
  
 #region BusinessScaleRange  
  
 public async Task<IList<BusinessScaleRange>> GetBusinessScaleRangesAsync()  
 {  
 return await \_cache.GetAsync(BusinessScaleCacheKey, async () =>  
 {  
 return await \_businessScaleRepo.Table  
 .OrderBy(x => x.MinValue)  
 .ToListAsync();  
 });  
 }  
  
 public async Task<BusinessScaleRange> GetBusinessScaleRangeByIdAsync(int id)  
 {  
 var list = await GetBusinessScaleRangesAsync();  
 return list.FirstOrDefault(x => x.Id == id);  
 }  
  
 public async Task<BusinessScaleRange> FindBusinessScaleRangeAsync(int employeesCount)  
 {  
 var list = await GetBusinessScaleRangesAsync();  
 return list.FirstOrDefault(x => employeesCount >= x.MinValue && employeesCount <= x.MaxValue);  
 }  
  
 #endregion  
  
 #region EmployeeRange  
  
 public async Task<IList<EmployeeRange>> GetEmployeeRangesAsync()  
 {  
 return await \_cache.GetAsync(EmployeeRangeCacheKey, async () =>  
 {  
 return await \_employeeRangeRepo.Table  
 .OrderBy(x => x.MinValue)  
 .ToListAsync();  
 });  
 }  
  
 public async Task<EmployeeRange> GetEmployeeRangeByIdAsync(int id)  
 {  
 var list = await GetEmployeeRangesAsync();  
 return list.FirstOrDefault(x => x.Id == id);  
 }  
  
 public async Task<EmployeeRange> FindEmployeeRangeAsync(int employeesCount)  
 {  
 var list = await GetEmployeeRangesAsync();  
 return list.FirstOrDefault(x => employeesCount >= x.MinValue && employeesCount <= x.MaxValue);  
 }  
  
 #endregion  
  
 #region Enums  
  
 public Task<IList<string>> GetLicenseTypesAsync()  
 {  
 var values = System.Enum.GetNames(typeof(LicenseType)).ToList();  
 return Task.FromResult<IList<string>>(values);  
 }  
  
 public Task<IList<string>> GetLicenseSectorsAsync()  
 {  
 var values = System.Enum.GetNames(typeof(LicenseSector)).ToList();  
 return Task.FromResult<IList<string>>(values);  
 }  
  
 public Task<IList<string>> GetFinancialDomainsAsync()  
 {  
 var values = System.Enum.GetNames(typeof(FinancialDomain)).ToList();  
 return Task.FromResult<IList<string>>(values);  
 }  
  
 #endregion  
  
 #region Cache Management  
  
 public async Task ClearLookupCacheAsync()  
 {  
 await \_cache.RemoveByPrefixAsync("Lookup");  
 }  
  
 #endregion  
 }  
}

## INotificationService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Notifications;  
  
namespace App.Services.Notifications  
{  
 public interface INotificationService  
 {  
 Task SendAsync(  
 int? registrationId,  
 NotificationEvent eventType,  
 int triggeredByUserId,  
 int recipientUserId,  
 NotificationChannel channel = NotificationChannel.InApp,  
 IDictionary<string, string> tokens = null);  
  
 Task<IList<NotificationLog>> GetLogsByUserAsync(int userId);  
 Task MarkAsReadAsync(int notificationId, int userId);  
 Task RetryFailedAsync(int maxAttempts = 3);  
 }  
}

## NotificationService.cs

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Notifications;  
using App.Core.RepositoryServices;  
using App.Services.Localization;  
using App.Services.Templates;  
using Microsoft.EntityFrameworkCore;  
  
namespace App.Services.Notifications  
{  
 public class NotificationService : INotificationService  
 {  
 private readonly IRepository<Notification> \_notificationRepository;  
 private readonly IRepository<NotificationLog> \_logRepository;  
 private readonly IRepository<NotificationReadLog> \_readLogRepository;  
 private readonly ILocalizationService \_localizationService;  
 private readonly ITemplateService \_templateService;  
  
 public NotificationService(  
 IRepository<Notification> notificationRepository,  
 IRepository<NotificationLog> logRepository,  
 IRepository<NotificationReadLog> readLogRepository,  
 ILocalizationService localizationService,  
 ITemplateService templateService)  
 {  
 \_notificationRepository = notificationRepository;  
 \_logRepository = logRepository;  
 \_readLogRepository = readLogRepository;  
 \_localizationService = localizationService;  
 \_templateService = templateService;  
 }  
  
 #region Sending  
  
 public async Task SendAsync(  
 int? registrationId,  
 NotificationEvent eventType,  
 int triggeredByUserId,  
 int recipientUserId,  
 NotificationChannel channel = NotificationChannel.InApp,  
 IDictionary<string, string> tokens = null)  
 {  
 // 1) Get template based on event + channel  
 var templates = await \_templateService.GetAllAsync(channel.ToString());  
 var template = templates.Data  
 .FirstOrDefault(t => t.Name.Equals($"Notification.{eventType}", StringComparison.OrdinalIgnoreCase));  
  
 // 2) Localize event  
 var eventName = await \_localizationService.GetLocalizedEnumAsync(eventType);  
  
 // 3) Build message  
 string message;  
 if (template != null)  
 {  
 message = template.Content;  
 if (tokens != null)  
 {  
 foreach (var kv in tokens)  
 message = message.Replace($"%{kv.Key}%", kv.Value);  
 }  
 }  
 else  
 {  
 var defaultTemplate = await \_localizationService.GetResourceAsync($"Notification.{eventType}");  
 message = \_localizationService.FormatMessage(defaultTemplate, registrationId, triggeredByUserId);  
 }  
  
 // 4) Save notification  
 var notification = new Notification  
 {  
 RegistrationId = registrationId,  
 EventType = eventType,  
 EventTypeId = (int)eventType,  
 RecipientUserId = recipientUserId,  
 TriggeredByUserId = triggeredByUserId,  
 Message = message,  
 ChannelId = (int)channel,  
 Channel = channel,  
 StatusId = (int)NotificationDeliveryStatus.Pending,  
 Status = NotificationDeliveryStatus.Pending,  
 CreatedOnUtc = DateTime.UtcNow  
 };  
  
 await \_notificationRepository.InsertAsync(notification);  
  
 // 5) Log  
 var log = new NotificationLog  
 {  
 NotificationId = notification.Id,  
 Channel = channel.ToString(),  
 SentOnUtc = DateTime.UtcNow  
 };  
  
 try  
 {  
 switch (channel)  
 {  
 case NotificationChannel.Email:  
 await SendEmailAsync(notification);  
 break;  
 case NotificationChannel.SMS:  
 await SendSmsAsync(notification);  
 break;  
 case NotificationChannel.Push:  
 await SendPushAsync(notification);  
 break;  
 case NotificationChannel.InApp:  
 default:  
 // InApp stored only  
 break;  
 }  
  
 notification.Status = NotificationDeliveryStatus.Sent;  
 notification.StatusId = (int)NotificationDeliveryStatus.Sent;  
 log.Success = true;  
 log.Response = "Delivered successfully";  
 }  
 catch (Exception ex)  
 {  
 notification.Status = NotificationDeliveryStatus.Failed;  
 notification.StatusId = (int)NotificationDeliveryStatus.Failed;  
 log.Success = false;  
 log.Response = ex.Message;  
 }  
  
 await \_notificationRepository.UpdateAsync(notification);  
 await \_logRepository.InsertAsync(log);  
 }  
  
 #endregion  
  
 #region Logs & ReadLogs  
  
 public async Task<IList<NotificationLog>> GetLogsByUserAsync(int userId)  
 {  
 var notifications = await \_notificationRepository.Table  
 .Where(n => n.RecipientUserId == userId)  
 .Select(n => n.Id)  
 .ToListAsync();  
  
 return await \_logRepository.Table  
 .Where(l => notifications.Contains(l.NotificationId))  
 .OrderByDescending(l => l.SentOnUtc)  
 .ToListAsync();  
 }  
  
 public async Task MarkAsReadAsync(int notificationId, int userId)  
 {  
 var entry = new NotificationReadLog  
 {  
 NotificationId = notificationId,  
 UserId = userId,  
 ReadOnUtc = DateTime.UtcNow  
 };  
  
 await \_readLogRepository.InsertAsync(entry);  
 }  
  
 #endregion  
  
 #region Retry  
  
 public async Task RetryFailedAsync(int maxAttempts = 3)  
 {  
 var failed = await \_notificationRepository.Table  
 .Where(n => n.Status == NotificationDeliveryStatus.Failed)  
 .Take(50)  
 .ToListAsync();  
  
 foreach (var notification in failed)  
 {  
 await SendAsync(  
 notification.RegistrationId,  
 notification.EventType,  
 notification.TriggeredByUserId,  
 notification.RecipientUserId,  
 notification.Channel,  
 new Dictionary<string, string> { { "Message", notification.Message } }  
 );  
 }  
 }  
  
 #endregion  
  
 #region Channels  
  
 private Task SendEmailAsync(Notification notification)  
 {  
 Console.WriteLine($"[EMAIL] To: {notification.RecipientUserId}, Msg: {notification.Message}");  
 return Task.CompletedTask;  
 }  
  
 private Task SendSmsAsync(Notification notification)  
 {  
 Console.WriteLine($"[SMS] To: {notification.RecipientUserId}, Msg: {notification.Message}");  
 return Task.CompletedTask;  
 }  
  
 private Task SendPushAsync(Notification notification)  
 {  
 Console.WriteLine($"[PUSH] To: {notification.RecipientUserId}, Msg: {notification.Message}");  
 return Task.CompletedTask;  
 }  
  
 #endregion  
 }  
}

## ContactService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Registrations  
{  
 public class ContactService : BaseService, IContactService  
 {  
 private readonly IRepository<FIContact> \_contactRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public ContactService(  
 IRepository<FIContact> contactRepository,  
 ILocalizationService localizationService)  
 {  
 \_contactRepository = contactRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<FIContact> GetByIdAsync(int id)  
 => await \_contactRepository.GetByIdAsync(id);  
  
 public async Task<IList<FIContact>> GetByRegistrationIdAsync(int registrationId)  
 => await \_contactRepository.GetAllAsync(q => q.Where(c => c.RegistrationId == registrationId));  
  
 public async Task<ServiceResult<FIContact>> InsertAsync(FIContact contact)  
 {  
 if (contact == null)  
 return Failed<FIContact>(await \_localizationService.GetResourceAsync("Contact.Insert.Null"));  
  
 await \_contactRepository.InsertAsync(contact);  
 return Success(contact, await \_localizationService.GetResourceAsync("Contact.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<FIContact>> UpdateAsync(FIContact contact)  
 {  
 if (contact == null)  
 return Failed<FIContact>(await \_localizationService.GetResourceAsync("Contact.Update.Null"));  
  
 await \_contactRepository.UpdateAsync(contact);  
 return Success(contact, await \_localizationService.GetResourceAsync("Contact.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var contact = await \_contactRepository.GetByIdAsync(id);  
 if (contact == null)  
 return Failed(await \_localizationService.GetResourceAsync("Contact.NotFound"));  
  
 await \_contactRepository.DeleteAsync(contact);  
 return Success(await \_localizationService.GetResourceAsync("Contact.Delete.Success"));  
 }  
 }  
}

## DocumentService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Registrations  
{  
 public class DocumentService : BaseService, IDocumentService  
 {  
 private readonly IRepository<FIDocument> \_documentRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public DocumentService(  
 IRepository<FIDocument> documentRepository,  
 ILocalizationService localizationService)  
 {  
 \_documentRepository = documentRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<FIDocument> GetByIdAsync(int id)  
 => await \_documentRepository.GetByIdAsync(id);  
  
 public async Task<IList<FIDocument>> GetByRegistrationIdAsync(int registrationId)  
 => await \_documentRepository.GetAllAsync(q => q.Where(d => d.RegistrationId == registrationId));  
  
 public async Task<ServiceResult<FIDocument>> InsertAsync(FIDocument document)  
 {  
 if (document == null)  
 return Failed<FIDocument>(await \_localizationService.GetResourceAsync("Document.Insert.Null"));  
  
 await \_documentRepository.InsertAsync(document);  
 return Success(document, await \_localizationService.GetResourceAsync("Document.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<FIDocument>> UpdateAsync(FIDocument document)  
 {  
 if (document == null)  
 return Failed<FIDocument>(await \_localizationService.GetResourceAsync("Document.Update.Null"));  
  
 await \_documentRepository.UpdateAsync(document);  
 return Success(document, await \_localizationService.GetResourceAsync("Document.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var document = await \_documentRepository.GetByIdAsync(id);  
 if (document == null)  
 return Failed(await \_localizationService.GetResourceAsync("Document.NotFound"));  
  
 await \_documentRepository.DeleteAsync(document);  
 return Success(await \_localizationService.GetResourceAsync("Document.Delete.Success"));  
 }  
 }  
}

## IContactService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
  
namespace App.Services.Registrations  
{  
 public interface IContactService  
 {  
 Task<FIContact> GetByIdAsync(int id);  
 Task<IList<FIContact>> GetByRegistrationIdAsync(int registrationId);  
 Task<ServiceResult<FIContact>> InsertAsync(FIContact contact);  
 Task<ServiceResult<FIContact>> UpdateAsync(FIContact contact);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## IDocumentService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
  
namespace App.Services.Registrations  
{  
 public interface IDocumentService  
 {  
 Task<FIDocument> GetByIdAsync(int id);  
 Task<IList<FIDocument>> GetByRegistrationIdAsync(int registrationId);  
  
 Task<ServiceResult<FIDocument>> InsertAsync(FIDocument document);  
 Task<ServiceResult<FIDocument>> UpdateAsync(FIDocument document);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## IRegistrationActionHintsService.cs

using App.Core.Domain.Registrations;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public interface IRegistrationActionHintsService  
 {  
 Task<IList<RegistrationAction>> GetAvailableActionsAsync(Registration registration, int currentUserId, string currentUserRole);  
 }  
}

## IRegistrationSearchService.cs

using App.Core.Domain.Ref;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
using System;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public interface IRegistrationSearchService  
 {  
 Task<PagedResult<Registration>> SearchAsync(  
 string institutionName = null,  
 string licenseNumber = null,  
 int? countryId = null,  
 RegistrationStatus? status = null,  
 LicenseType? licenseType = null,  
 LicenseSector? licenseSector = null,  
 FinancialDomain? financialDomain = null,  
 DateTime? createdFromUtc = null,  
 DateTime? createdToUtc = null,  
 int pageIndex = 0,  
 int pageSize = int.MaxValue  
 );  
 }  
}

## IRegistrationService.cs

using App.Core.Domain.Notifications;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public interface IRegistrationService  
 {  
 #region CRUD  
  
 Task<Registration> GetByIdAsync(int id);  
 Task<IList<Registration>> GetAllAsync();  
 Task<ServiceResult<Registration>> InsertAsync(Registration registration);  
 Task<ServiceResult<Registration>> UpdateAsync(Registration registration);  
 Task<ServiceResult> DeleteAsync(int id);  
  
 #endregion  
  
 #region Workflow  
  
 Task<ServiceResult> SubmitAsync(int registrationId, int performedByUserId, string remarks = null);  
 Task<ServiceResult> ValidateAsync(int registrationId, int performedByUserId, ValidationStatus status, string remarks = null);  
 Task<ServiceResult> ApproveAsync(int registrationId, int performedByUserId, ApprovalStatus status, string remarks = null);  
 Task<ServiceResult> AuditAsync(int registrationId, int performedByUserId, AuditStatus status, string remarks = null);  
 Task<ServiceResult> ReturnForEditAsync(int registrationId, int performedByUserId, string remarks = null);  
 Task<ServiceResult> RejectAsync(int registrationId, int performedByUserId, string remarks = null);  
 Task<ServiceResult> ArchiveAsync(int registrationId, int performedByUserId, string remarks = null);  
 Task<ServiceResult> FinalSubmissionAsync(int registrationId, int performedByUserId, string remarks = null);  
  
 #endregion  
  
 #region Contacts & Documents  
  
 Task<IList<FIContact>> GetContactsByRegistrationIdAsync(int registrationId);  
 Task<IList<FIDocument>> GetDocumentsByRegistrationIdAsync(int registrationId);  
 Task<ServiceResult> AddContactAsync(int registrationId, FIContact contact);  
 Task<ServiceResult> AddDocumentAsync(int registrationId, FIDocument document);  
  
 #endregion  
 #region Validation (New)  
 /// <summary>  
 /// Checks whether a registration already exists with the same institution name or license number.  
 /// </summary>  
 Task<bool> CheckDuplicateAsync(string institutionName, string licenseNumber);  
 #endregion  
  
 #region Status & Audit (New)  
 /// <summary>  
 /// Returns the full status change history (audit log) for a given registration.  
 /// </summary>  
 Task<IList<FIRegistrationStatusLog>> GetStatusHistoryAsync(int registrationId);  
 #endregion  
  
 #region Notifications (New)  
 /// <summary>  
 /// Sends notification for workflow events (submit, approve, reject, return, etc.).  
 /// </summary>  
 Task NotifyAsync(int registrationId, NotificationEvent eventType, int triggeredByUserId, int recipientUserId, NotificationChannel channel = NotificationChannel.InApp, string customMessage = null);  
 #endregion  
 }  
}

## IRegistrationStatusLogService.cs

﻿using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
  
namespace App.Services.Registrations  
{  
 public interface IRegistrationStatusLogService  
 {  
 Task<FIRegistrationStatusLog> GetByIdAsync(int id);  
 Task<IList<FIRegistrationStatusLog>> GetByRegistrationIdAsync(int registrationId);  
 Task<ServiceResult<FIRegistrationStatusLog>> InsertAsync(FIRegistrationStatusLog log);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## IRegistrationWorkflowService.cs

using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
  
namespace App.Services.Registrations  
{  
 public interface IRegistrationWorkflowService  
 {  
 Task<ServiceResult> SubmitAsync(int registrationId, int performedByUserId, int recipientUserId);  
 Task<ServiceResult> ValidateAsync(int registrationId, int performedByUserId, int recipientUserId, ValidationStatus status);  
 Task<ServiceResult> ApproveAsync(int registrationId, int performedByUserId, int recipientUserId, ApprovalStatus status);  
 Task<ServiceResult> AuditAsync(int registrationId, int performedByUserId, int recipientUserId, AuditStatus status);  
 Task<ServiceResult> ReturnForEditAsync(int registrationId, int performedByUserId, int recipientUserId);  
 Task<ServiceResult> RejectAsync(int registrationId, int performedByUserId, int recipientUserId);  
 Task<ServiceResult> ArchiveAsync(int registrationId, int performedByUserId, int recipientUserId);  
 Task<ServiceResult> FinalSubmissionAsync(int registrationId, int performedByUserId, int recipientUserId);  
 }  
}

## RegistrationActionHintsService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
  
namespace App.Services.Registrations  
{  
 public class RegistrationActionHintsService : IRegistrationActionHintsService  
 {  
 public Task<IList<RegistrationAction>> GetAvailableActionsAsync(Registration registration, int currentUserId, string currentUserRole)  
 {  
 var actions = new List<RegistrationAction>();  
  
 switch (registration.Status)  
 {  
 case RegistrationStatus.Draft:  
 if (currentUserRole == "Maker")  
 actions.Add(RegistrationAction.Submit);  
 break;  
  
 case RegistrationStatus.Submitted:  
 if (currentUserRole == "Checker")  
 actions.Add(RegistrationAction.Validate);  
 break;  
  
 case RegistrationStatus.UnderReview:  
 if (currentUserRole == "Checker")  
 actions.Add(RegistrationAction.Validate);  
 if (currentUserRole == "Approver")  
 actions.Add(RegistrationAction.Approve);  
 break;  
  
 case RegistrationStatus.Approved:  
 if (currentUserRole == "Regulator")  
 actions.Add(RegistrationAction.Audit);  
 break;  
  
 case RegistrationStatus.ReturnedForEdit:  
 if (currentUserRole == "Maker")  
 actions.Add(RegistrationAction.Submit);  
 break;  
  
 case RegistrationStatus.Rejected:  
 if (currentUserRole == "Maker")  
 actions.Add(RegistrationAction.Submit);  
 break;  
  
 case RegistrationStatus.FinalSubmission:  
 if (currentUserRole == "Admin")  
 actions.Add(RegistrationAction.Archive);  
 break;  
  
 case RegistrationStatus.Archived:  
 break;  
 }  
  
 return Task.FromResult<IList<RegistrationAction>>(actions);  
 }  
 }  
}

## RegistrationSearchService.cs

using App.Core.Domain.Ref;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using System;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public class RegistrationSearchService : IRegistrationSearchService  
 {  
 private readonly IRepository<Registration> \_registrationRepository;  
  
 public RegistrationSearchService(IRepository<Registration> registrationRepository)  
 {  
 \_registrationRepository = registrationRepository;  
 }  
  
 public async Task<Common.PagedResult<Registration>> SearchAsync(  
 string institutionName = null,  
 string licenseNumber = null,  
 int? countryId = null,  
 RegistrationStatus? status = null,  
 LicenseType? licenseType = null,  
 LicenseSector? licenseSector = null,  
 FinancialDomain? financialDomain = null,  
 DateTime? createdFromUtc = null,  
 DateTime? createdToUtc = null,  
 int pageIndex = 0,  
 int pageSize = int.MaxValue)  
 {  
 var query = \_registrationRepository.Table;  
  
 if (!string.IsNullOrEmpty(institutionName))  
 query = query.Where(r => r.InstitutionName.Contains(institutionName));  
  
 if (!string.IsNullOrEmpty(licenseNumber))  
 query = query.Where(r => r.LicenseNumber.Contains(licenseNumber));  
  
 if (countryId.HasValue)  
 query = query.Where(r => r.CountryId == countryId);  
  
 if (status.HasValue)  
 query = query.Where(r => r.Status == status);  
  
 if (licenseType.HasValue)  
 query = query.Where(r => r.LicenseType == licenseType);  
  
 if (licenseSector.HasValue)  
 query = query.Where(r => r.LicenseSector == licenseSector);  
  
 if (financialDomain.HasValue)  
 query = query.Where(r => r.FinancialDomain == financialDomain);  
  
 if (createdFromUtc.HasValue)  
 query = query.Where(r => r.CreatedOnUtc >= createdFromUtc.Value);  
  
 if (createdToUtc.HasValue)  
 query = query.Where(r => r.CreatedOnUtc <= createdToUtc.Value);  
  
 var totalCount = query.Count();  
  
 var items = query  
 .OrderByDescending(r => r.CreatedOnUtc)  
 .Skip(pageIndex \* pageSize)  
 .Take(pageSize)  
 .ToList();  
  
 return await Task.FromResult(new Common.PagedResult<Registration>(items, totalCount, pageIndex, pageSize));  
 }  
 }  
}

## RegistrationService.cs

using App.Core.Domain.Notifications;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
using App.Services.Notifications;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public class RegistrationService : BaseService, IRegistrationService  
 {  
 private readonly IRepository<Registration> \_registrationRepository;  
 private readonly IRepository<FIContact> \_fiContactRepository;  
 private readonly IRepository<FIDocument> \_fiDocumentRepository;  
 private readonly ILocalizationService \_localizationService;  
 private readonly IRegistrationStatusLogService \_statusLogService;  
 private readonly INotificationService \_notificationService;  
  
 public RegistrationService(  
 IRepository<Registration> registrationRepository,  
 IRepository<FIContact> fiContactRepository,  
 IRepository<FIDocument> fiDocumentRepository,  
 ILocalizationService localizationService,  
 IRegistrationStatusLogService statusLogService,  
 INotificationService notificationService)  
 {  
 \_registrationRepository = registrationRepository;  
 \_fiContactRepository = fiContactRepository;  
 \_fiDocumentRepository = fiDocumentRepository;  
 \_localizationService = localizationService;  
 \_statusLogService = statusLogService;  
 \_notificationService = notificationService;  
 }  
  
 #region CRUD  
  
 public async Task<Registration> GetByIdAsync(int id)  
 => await \_registrationRepository.GetByIdAsync(id);  
  
 public async Task<IList<Registration>> GetAllAsync()  
 => await \_registrationRepository.GetAllAsync(q => q.OrderBy(r => r.CreatedOnUtc));  
  
 public async Task<ServiceResult<Registration>> InsertAsync(Registration registration)  
 {  
 if (registration == null)  
 return Failed<Registration>(await \_localizationService.GetResourceAsync("Registration.Insert.Null"));  
  
 registration.CreatedOnUtc = DateTime.UtcNow;  
 await \_registrationRepository.InsertAsync(registration);  
  
 // Localized log  
 var logTemplate = await \_localizationService.GetResourceAsync("Registration.Created.Log");  
 var logMessage = \_localizationService.FormatMessage(logTemplate, registration.CreatedByUserId);  
  
 var log = new FIRegistrationStatusLog  
 {  
 RegistrationId = registration.Id,  
 RegistrationStatus = registration.Status,  
 PerformedBy = registration.CreatedByUserId,  
 ActionDateUtc = DateTime.UtcNow,  
 Remarks = logMessage  
 };  
 await \_statusLogService.InsertAsync(log);  
  
 var successMessage = await \_localizationService.GetResourceAsync("Registration.Insert.Success");  
 return Success(registration, successMessage);  
 }  
  
 public async Task<ServiceResult<Registration>> UpdateAsync(Registration registration)  
 {  
 if (registration == null)  
 return Failed<Registration>(await \_localizationService.GetResourceAsync("Registration.Update.Null"));  
  
 await \_registrationRepository.UpdateAsync(registration);  
  
 var successMessage = await \_localizationService.GetResourceAsync("Registration.Update.Success");  
 return Success(registration, successMessage);  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var registration = await \_registrationRepository.GetByIdAsync(id);  
 if (registration == null)  
 return Failed(await \_localizationService.GetResourceAsync("Registration.NotFound"));  
  
 await \_registrationRepository.DeleteAsync(registration);  
  
 var successMessage = await \_localizationService.GetResourceAsync("Registration.Delete.Success");  
 return Success(successMessage);  
 }  
  
 #endregion  
  
 #region Workflow  
  
 public async Task<ServiceResult> SubmitAsync(int registrationId, int performedByUserId, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.Submitted, performedByUserId, remarks);  
  
 public async Task<ServiceResult> ValidateAsync(int registrationId, int performedByUserId, ValidationStatus status, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.UnderReview, performedByUserId, remarks, validationStatus: status);  
  
 public async Task<ServiceResult> ApproveAsync(int registrationId, int performedByUserId, ApprovalStatus status, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.Approved, performedByUserId, remarks, approvalStatus: status);  
  
 public async Task<ServiceResult> AuditAsync(int registrationId, int performedByUserId, AuditStatus status, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.UnderReview, performedByUserId, remarks, auditStatus: status);  
  
 public async Task<ServiceResult> ReturnForEditAsync(int registrationId, int performedByUserId, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.ReturnedForEdit, performedByUserId, remarks);  
  
 public async Task<ServiceResult> RejectAsync(int registrationId, int performedByUserId, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.Rejected, performedByUserId, remarks);  
  
 public async Task<ServiceResult> ArchiveAsync(int registrationId, int performedByUserId, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.Archived, performedByUserId, remarks);  
  
 public async Task<ServiceResult> FinalSubmissionAsync(int registrationId, int performedByUserId, string remarks = null)  
 => await ChangeStatusAsync(registrationId, RegistrationStatus.FinalSubmission, performedByUserId, remarks);  
  
 private async Task<ServiceResult> ChangeStatusAsync(  
 int registrationId,  
 RegistrationStatus newStatus,  
 int performedByUserId,  
 string remarks,  
 ValidationStatus? validationStatus = null,  
 ApprovalStatus? approvalStatus = null,  
 AuditStatus? auditStatus = null)  
 {  
 var registration = await \_registrationRepository.GetByIdAsync(registrationId);  
 if (registration == null)  
 return Failed(await \_localizationService.GetResourceAsync("Registration.NotFound"));  
  
 var oldStatus = registration.Status;  
 registration.Status = newStatus;  
 await \_registrationRepository.UpdateAsync(registration);  
  
 // Localize enums  
 var oldStatusName = await \_localizationService.GetLocalizedEnumAsync(oldStatus);  
 var newStatusName = await \_localizationService.GetLocalizedEnumAsync(newStatus);  
  
 // Localized log  
 var logTemplate = await \_localizationService.GetResourceAsync("Registration.Status.Changed");  
 var logMessage = \_localizationService.FormatMessage(logTemplate, oldStatusName, newStatusName);  
  
 var log = new FIRegistrationStatusLog  
 {  
 RegistrationId = registration.Id,  
 RegistrationStatus = newStatus,  
 ValidationStatus = validationStatus,  
 ApprovalStatus = approvalStatus,  
 AuditStatus = auditStatus,  
 PerformedBy = performedByUserId,  
 ActionDateUtc = DateTime.UtcNow,  
 Remarks = remarks ?? logMessage  
 };  
 await \_statusLogService.InsertAsync(log);  
  
 var successTemplate = await \_localizationService.GetResourceAsync("Registration.Status.Success");  
 var successMessage = \_localizationService.FormatMessage(successTemplate, newStatusName);  
  
 return Success(successMessage);  
 }  
  
 #endregion  
  
 #region Contacts & Documents  
  
 public async Task<IList<FIContact>> GetContactsByRegistrationIdAsync(int registrationId)  
 => await \_fiContactRepository.GetAllAsync(q => q.Where(c => c.RegistrationId == registrationId));  
  
 public async Task<IList<FIDocument>> GetDocumentsByRegistrationIdAsync(int registrationId)  
 => await \_fiDocumentRepository.GetAllAsync(q => q.Where(d => d.RegistrationId == registrationId));  
  
 public async Task<ServiceResult> AddContactAsync(int registrationId, FIContact contact)  
 {  
 if (contact == null)  
 return Failed(await \_localizationService.GetResourceAsync("Registration.Contact.Null"));  
  
 contact.RegistrationId = registrationId;  
 await \_fiContactRepository.InsertAsync(contact);  
  
 return Success(await \_localizationService.GetResourceAsync("Registration.Contact.Added"));  
 }  
  
 public async Task<ServiceResult> AddDocumentAsync(int registrationId, FIDocument document)  
 {  
 if (document == null)  
 return Failed(await \_localizationService.GetResourceAsync("Registration.Document.Null"));  
  
 document.RegistrationId = registrationId;  
 await \_fiDocumentRepository.InsertAsync(document);  
  
 return Success(await \_localizationService.GetResourceAsync("Registration.Document.Added"));  
 }  
  
 public async Task<bool> CheckDuplicateAsync(string institutionName, string licenseNumber)  
 {  
 var duplicates = await \_registrationRepository.GetAllAsync(q =>  
 q.Where(s => s.InstitutionName == institutionName || s.LicenseNumber == licenseNumber));  
 return duplicates != null && duplicates.Count > 0;  
 }  
  
 #endregion  
  
 #region Status & Audit (New)  
  
 public async Task<IList<FIRegistrationStatusLog>> GetStatusHistoryAsync(int registrationId)  
 {  
 return await \_statusLogService.GetByRegistrationIdAsync(registrationId);  
 }  
  
 public async Task NotifyAsync(int registrationId, NotificationEvent eventType, int triggeredByUserId, int recipientUserId, NotificationChannel channel = NotificationChannel.InApp, string customMessage = null)  
 {  
 await \_notificationService.SendAsync(  
 registrationId,  
 eventType,  
 triggeredByUserId,  
 recipientUserId,  
 channel,  
 customMessage  
 );  
 }  
  
  
 #endregion  
 }  
}

## RegistrationStatusLogService.cs

﻿using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Registrations;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Registrations  
{  
 public class RegistrationStatusLogService : BaseService, IRegistrationStatusLogService  
 {  
 private readonly IRepository<FIRegistrationStatusLog> \_statusLogRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public RegistrationStatusLogService(  
 IRepository<FIRegistrationStatusLog> statusLogRepository,  
 ILocalizationService localizationService)  
 {  
 \_statusLogRepository = statusLogRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<FIRegistrationStatusLog> GetByIdAsync(int id)  
 => await \_statusLogRepository.GetByIdAsync(id);  
  
 public async Task<IList<FIRegistrationStatusLog>> GetByRegistrationIdAsync(int registrationId)  
 => await \_statusLogRepository.GetAllAsync(q => q.Where(l => l.RegistrationId == registrationId)  
 .OrderByDescending(l => l.ActionDateUtc));  
  
 public async Task<ServiceResult<FIRegistrationStatusLog>> InsertAsync(FIRegistrationStatusLog log)  
 {  
 if (log == null)  
 return Failed<FIRegistrationStatusLog>(await \_localizationService.GetResourceAsync("StatusLog.Insert.Null"));  
  
 await \_statusLogRepository.InsertAsync(log);  
 return Success(log, await \_localizationService.GetResourceAsync("StatusLog.Insert.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var log = await \_statusLogRepository.GetByIdAsync(id);  
 if (log == null)  
 return Failed(await \_localizationService.GetResourceAsync("StatusLog.NotFound"));  
  
 await \_statusLogRepository.DeleteAsync(log);  
 return Success(await \_localizationService.GetResourceAsync("StatusLog.Delete.Success"));  
 }  
 }  
}

## RegistrationWorkflowService.cs

using App.Core.Domain.Notifications;  
using App.Core.Domain.Registrations;  
using App.Services.Common;  
using App.Services.Localization;  
using App.Services.Notifications;  
using System.Threading.Tasks;  
  
namespace App.Services.Registrations  
{  
 public class RegistrationWorkflowService : BaseService, IRegistrationWorkflowService  
 {  
 private readonly IRegistrationService \_registrationService;  
 private readonly INotificationService \_notificationService;  
 private readonly ILocalizationService \_localizationService;  
  
 public RegistrationWorkflowService(  
 IRegistrationService registrationService,  
 INotificationService notificationService,  
 ILocalizationService localizationService)  
 {  
 \_registrationService = registrationService;  
 \_notificationService = notificationService;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult> SubmitAsync(int registrationId, int performedByUserId, int recipientUserId)  
 {  
 var result = await \_registrationService.SubmitAsync(registrationId, performedByUserId);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationSubmitted, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Submit.Success"));  
 }  
  
 public async Task<ServiceResult> ValidateAsync(int registrationId, int performedByUserId, int recipientUserId, ValidationStatus status)  
 {  
 var result = await \_registrationService.ValidateAsync(registrationId, performedByUserId, status);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationValidated, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Validate.Success"));  
 }  
  
 public async Task<ServiceResult> ApproveAsync(int registrationId, int performedByUserId, int recipientUserId, ApprovalStatus status)  
 {  
 var result = await \_registrationService.ApproveAsync(registrationId, performedByUserId, status);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationApproved, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Approve.Success"));  
 }  
  
 public async Task<ServiceResult> AuditAsync(int registrationId, int performedByUserId, int recipientUserId, AuditStatus status)  
 {  
 var result = await \_registrationService.AuditAsync(registrationId, performedByUserId, status);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationAudited, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Audit.Success"));  
 }  
  
 public async Task<ServiceResult> ReturnForEditAsync(int registrationId, int performedByUserId, int recipientUserId)  
 {  
 var result = await \_registrationService.ReturnForEditAsync(registrationId, performedByUserId);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationReturnedForEdit, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.ReturnForEdit.Success"));  
 }  
  
 public async Task<ServiceResult> RejectAsync(int registrationId, int performedByUserId, int recipientUserId)  
 {  
 var result = await \_registrationService.RejectAsync(registrationId, performedByUserId);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationRejected, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Reject.Success"));  
 }  
  
 public async Task<ServiceResult> ArchiveAsync(int registrationId, int performedByUserId, int recipientUserId)  
 {  
 var result = await \_registrationService.ArchiveAsync(registrationId, performedByUserId);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationArchived, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.Archive.Success"));  
 }  
  
 public async Task<ServiceResult> FinalSubmissionAsync(int registrationId, int performedByUserId, int recipientUserId)  
 {  
 var result = await \_registrationService.FinalSubmissionAsync(registrationId, performedByUserId);  
 if (!result.Success) return result;  
  
 await \_notificationService.SendAsync(registrationId, NotificationEvent.RegistrationFinalSubmission, performedByUserId, recipientUserId);  
 return Success(await \_localizationService.GetResourceAsync("Workflow.FinalSubmission.Success"));  
 }  
 }  
}

## IReportService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Reports;  
using App.Services.Common;  
  
namespace App.Services.Reports  
{  
 public interface IReportService  
 {  
 Task<ServiceResult<Report>> GetAsync(int id);  
 Task<ServiceResult<IReadOnlyList<Report>>> GetAllAsync(string type = null);  
 Task<ServiceResult<Report>> CreateAsync(Report model);  
 Task<ServiceResult> UpdateAsync(int id, Report model);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## ReportService.cs

using System.Linq;  
using System.Threading.Tasks;  
using System.Collections.Generic;  
using App.Core.Domain.Reports;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Reports  
{  
 public class ReportService : BaseService, IReportService  
 {  
 private readonly IRepository<Report> \_reportRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public ReportService(IRepository<Report> reportRepository, ILocalizationService localizationService)  
 {  
 \_reportRepository = reportRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult<Report>> GetAsync(int id)  
 {  
 var entity = await \_reportRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult<Report>.Failed(await \_localizationService.GetResourceAsync("Report.NotFound"));  
 return ServiceResult<Report>.Success(entity);  
 }  
  
 public async Task<ServiceResult<IReadOnlyList<Report>>> GetAllAsync(string type = null)  
 {  
 var list = await \_reportRepository.GetAllAsync(q =>  
 {  
 if (!string.IsNullOrWhiteSpace(type)) q = q.Where(x => x.ReportType == type);  
 return q;  
 });  
 return ServiceResult<IReadOnlyList<Report>>.Success(list.ToList());  
 }  
  
 public async Task<ServiceResult<Report>> CreateAsync(Report model)  
 {  
 if (model == null) return ServiceResult<Report>.Failed(await \_localizationService.GetResourceAsync("Errors.NullModel"));  
 await \_reportRepository.InsertAsync(model);  
 return ServiceResult<Report>.Success(model);  
 }  
  
 public async Task<ServiceResult> UpdateAsync(int id, Report model)  
 {  
 var entity = await \_reportRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Report.NotFound"));  
 entity.Title = model.Title;  
 entity.ReportType = model.ReportType;  
 entity.FilePath = model.FilePath;  
 await \_reportRepository.UpdateAsync(entity);  
 return ServiceResult.SuccessResult();  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var entity = await \_reportRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Report.NotFound"));  
 await \_reportRepository.DeleteAsync(entity);  
 return ServiceResult.SuccessResult();  
 }  
 }  
}

## AccessControlService.cs

using System.Threading.Tasks;  
using App.Core.Security;  
  
namespace App.Services.Security  
{  
 public class AccessControlService : IAccessControlService  
 {  
 private readonly IPermissionService \_permissionService;  
  
 public AccessControlService(IPermissionService permissionService)  
 {  
 \_permissionService = permissionService;  
 }  
  
 public async Task<bool> CanAsync(int userId, string entity, CrudAction action)  
 {  
 string permission = action switch  
 {  
 CrudAction.Create => $"{entity}.Create",  
 CrudAction.Read => $"{entity}.Read",  
 CrudAction.Update => $"{entity}.Update",  
 CrudAction.Delete => $"{entity}.Delete",  
 \_ => null  
 };  
  
 if (string.IsNullOrEmpty(permission))  
 return false;  
  
 return await \_permissionService.AuthorizeAsync(userId, permission);  
 }  
  
 public async Task<bool> CanDoAsync(int userId, string permissionSystemName)  
 {  
 return await \_permissionService.AuthorizeAsync(userId, permissionSystemName);  
 }  
 }  
}

## AuthenticationService.cs

using System;  
using System.Threading.Tasks;  
using App.Core.Domain.Users;  
using App.Services.Security;  
using App.Services.Users;  
using Microsoft.Extensions.Configuration;  
using Microsoft.IdentityModel.Tokens;  
using System.IdentityModel.Tokens.Jwt;  
using System.Text;  
using System.Security.Claims;  
  
namespace App.Services.Authentication  
{  
 public class AuthenticationService : IAuthenticationService  
 {  
 private readonly IUserService \_userService;  
 private readonly IEncryptionService \_encryptionService;  
 private readonly IConfiguration \_configuration;  
  
 public AuthenticationService(  
 IUserService userService,  
 IEncryptionService encryptionService,  
 IConfiguration configuration)  
 {  
 \_userService = userService;  
 \_encryptionService = encryptionService;  
 \_configuration = configuration;  
 }  
  
 public async Task<User> ValidateUserAsync(string username, string password)  
 {  
 var user = await \_userService.GetByUsernameAsync(username);  
 if (user == null || !user.IsActive)  
 return null;  
  
 var valid = \_encryptionService.VerifyPassword(  
 password,  
 user.PasswordHash,  
 user.PasswordSalt,  
 user.PasswordFormat  
 );  
  
 return valid ? user : null;  
 }  
  
 public async Task<string> GenerateTokenAsync(User user)  
 {  
 var jwtKey = \_configuration["Jwt:Key"];  
 var jwtIssuer = \_configuration["Jwt:Issuer"];  
 var jwtExpiry = int.TryParse(\_configuration["Jwt:ExpiryMinutes"], out var exp) ? exp : 120;  
  
 var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey));  
 var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  
  
 var roles = await \_userService.GetRolesAsync(user.Id);  
  
 var claims = new[]  
 {  
 new Claim(JwtRegisteredClaimNames.Sub, user.Username),  
 new Claim("userId", user.Id.ToString()),  
 new Claim("roles", string.Join(",", roles))  
 };  
  
 var token = new JwtSecurityToken(  
 issuer: jwtIssuer,  
 audience: jwtIssuer,  
 claims: claims,  
 expires: DateTime.UtcNow.AddMinutes(jwtExpiry),  
 signingCredentials: creds);  
  
 return new JwtSecurityTokenHandler().WriteToken(token);  
 }  
  
 public Task LogoutAsync(int userId)  
 {  
 // ?? ?????? ? RefreshTokens: ???? ?? DB  
 // ?? ???? Blacklist ??? JWT  
 return Task.CompletedTask;  
 }  
  
 public Task<string> RefreshTokenAsync(string token)  
 {  
 // Validate old token and issue a new one  
 return Task.FromResult(token); // Placeholder, ???? ??????? ?????  
 }  
 }  
}

## EncryptionService.cs

using App.Core.Configuration;  
using App.Core.Domain.Users;  
using App.Core.Security;  
using Microsoft.Extensions.Options;  
using System;  
using System.IO;  
using System.Linq;  
using System.Security.Cryptography;  
using System.Text;  
  
namespace App.Services.Security  
{  
 /// <summary>  
 /// Production-grade crypto: PBKDF2 for passwords, AES-256-CBC for encryption.  
 /// </summary>  
 public class EncryptionService : IEncryptionService  
 {  
 private readonly SecuritySettings \_settings;  
 private readonly byte[] \_aesKey; // 32 bytes  
  
 public EncryptionService(IOptions<SecuritySettings> settings)  
 {  
 \_settings = settings.Value ?? new SecuritySettings();  
  
 // Derive a 256-bit key from passphrase using SHA256 (fast KDF for key material).  
 // Passphrase itself should be stored securely (KeyVault/Secrets).  
 using var sha = SHA256.Create();  
 \_aesKey = sha.ComputeHash(Encoding.UTF8.GetBytes(\_settings.EncryptionPassphrase ?? "CHANGE\_ME\_IN\_SECRETS"));  
 }  
  
 #region Passwords  
  
 public string CreateSaltKey(int size = 32)  
 {  
 var bytes = new byte[size];  
 using var rng = RandomNumberGenerator.Create();  
 rng.GetBytes(bytes);  
 return Convert.ToBase64String(bytes);  
 }  
  
 public string CreatePasswordHash(string password, string saltKey, PasswordFormat format = PasswordFormat.Hashed)  
 {  
 if (password == null) throw new ArgumentNullException(nameof(password));  
 if (saltKey == null) throw new ArgumentNullException(nameof(saltKey));  
  
 return format switch  
 {  
 PasswordFormat.Clear => password,  
 PasswordFormat.Encrypted => EncryptText(password),  
 PasswordFormat.Hashed => PBKDF2(password, saltKey, \_settings.PasswordHashIterations),  
 \_ => throw new NotSupportedException($"Unsupported password format: {format}")  
 };  
 }  
  
 public bool VerifyPassword(string enteredPassword, string storedHash, string storedSalt, PasswordFormat format)  
 {  
 if (enteredPassword == null) return false;  
  
 switch (format)  
 {  
 case PasswordFormat.Clear:  
 return enteredPassword == storedHash;  
  
 case PasswordFormat.Encrypted:  
 try  
 {  
 var plain = DecryptText(storedHash);  
 return enteredPassword == plain;  
 }  
 catch { return false; }  
  
 case PasswordFormat.Hashed:  
 var computed = PBKDF2(enteredPassword, storedSalt, \_settings.PasswordHashIterations);  
 // constant-time compare  
 return FixedTimeEquals(  
 Convert.FromBase64String(computed),  
 Convert.FromBase64String(storedHash));  
  
 default:  
 return false;  
 }  
 }  
  
 private static string PBKDF2(string password, string saltBase64, int iterations)  
 {  
 var salt = Convert.FromBase64String(saltBase64);  
 using var pbkdf2 = new Rfc2898DeriveBytes(password, salt, iterations, HashAlgorithmName.SHA256);  
 var key = pbkdf2.GetBytes(32); // 256-bit  
 return Convert.ToBase64String(key);  
 }  
  
 private static bool FixedTimeEquals(byte[] a, byte[] b)  
 {  
 if (a == null || b == null || a.Length != b.Length) return false;  
 int diff = 0;  
 for (int i = 0; i < a.Length; i++) diff |= a[i] ^ b[i];  
 return diff == 0;  
 }  
  
 #endregion  
  
 #region Tokens & Hashes  
  
 public string GenerateSecureToken(int size = 64)  
 {  
 var bytes = new byte[size];  
 using var rng = RandomNumberGenerator.Create();  
 rng.GetBytes(bytes);  
 // URL-safe Base64 without padding  
 return Convert.ToBase64String(bytes)  
 .TrimEnd('=')  
 .Replace('+', '-')  
 .Replace('/', '\_');  
 }  
  
 public string ComputeSha256(string input)  
 {  
 using var sha = SHA256.Create();  
 var hash = sha.ComputeHash(Encoding.UTF8.GetBytes(input ?? string.Empty));  
 return Convert.ToBase64String(hash);  
 }  
  
 public string ComputeHmac(string input, string key)  
 {  
 using var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(key ?? string.Empty));  
 var mac = hmac.ComputeHash(Encoding.UTF8.GetBytes(input ?? string.Empty));  
 return Convert.ToBase64String(mac);  
 }  
  
 #endregion  
  
 #region AES-256-CBC  
  
 public string EncryptText(string plainText)  
 {  
 if (plainText == null) return null;  
  
 using var aes = Aes.Create();  
 aes.Key = \_aesKey;  
 aes.Mode = CipherMode.CBC;  
 aes.Padding = PaddingMode.PKCS7;  
  
 aes.GenerateIV(); // 16 bytes  
 using var encryptor = aes.CreateEncryptor(aes.Key, aes.IV);  
  
 var plainBytes = Encoding.UTF8.GetBytes(plainText);  
 var cipherBytes = encryptor.TransformFinalBlock(plainBytes, 0, plainBytes.Length);  
  
 // store IV + cipher as base64  
 var payload = new byte[aes.IV.Length + cipherBytes.Length];  
 Buffer.BlockCopy(aes.IV, 0, payload, 0, aes.IV.Length);  
 Buffer.BlockCopy(cipherBytes, 0, payload, aes.IV.Length, cipherBytes.Length);  
  
 return Convert.ToBase64String(payload);  
 }  
  
 public string DecryptText(string cipherText)  
 {  
 if (cipherText == null) return null;  
  
 var payload = Convert.FromBase64String(cipherText);  
  
 using var aes = Aes.Create();  
 aes.Key = \_aesKey;  
 aes.Mode = CipherMode.CBC;  
 aes.Padding = PaddingMode.PKCS7;  
  
 var iv = new byte[16];  
 var cipherBytes = new byte[payload.Length - 16];  
 Buffer.BlockCopy(payload, 0, iv, 0, 16);  
 Buffer.BlockCopy(payload, 16, cipherBytes, 0, cipherBytes.Length);  
  
 using var decryptor = aes.CreateDecryptor(aes.Key, iv);  
 var plainBytes = decryptor.TransformFinalBlock(cipherBytes, 0, cipherBytes.Length);  
 return Encoding.UTF8.GetString(plainBytes);  
 }  
  
 #endregion  
 }  
}

## IAccessControlService.cs

using System.Threading.Tasks;  
  
namespace App.Services.Security  
{  
 public enum CrudAction { Create, Read, Update, Delete }  
  
 public interface IAccessControlService  
 {  
 Task<bool> CanAsync(int userId, string entity, CrudAction action);  
  
 Task<bool> CanDoAsync(int userId, string permissionSystemName);  
 }  
  
}

## IAuthenticationService.cs

using System.Threading.Tasks;  
using App.Core.Domain.Users;  
  
namespace App.Services.Authentication  
{  
 public interface IAuthenticationService  
 {  
 /// <summary>  
 /// Validate username & password, return user if success.  
 /// </summary>  
 Task<User> ValidateUserAsync(string username, string password);  
  
 /// <summary>  
 /// Generate JWT or session token for authenticated user.  
 /// </summary>  
 Task<string> GenerateTokenAsync(User user);  
  
 /// <summary>  
 /// Invalidate user session / token.  
 /// </summary>  
 Task LogoutAsync(int userId);  
  
 /// <summary>  
 /// Refresh JWT token (for long-running sessions).  
 /// </summary>  
 Task<string> RefreshTokenAsync(string token);  
 }  
}

## IEncryptionService.cs

using App.Core.Domain.Users;  
  
namespace App.Services.Security  
{  
 public interface IEncryptionService  
 {  
 /// <summary>  
 /// Create a hash for a given password with salt & format.  
 /// </summary>  
 string CreatePasswordHash(string password, string saltKey, PasswordFormat format = PasswordFormat.Hashed);  
  
 /// <summary>  
 /// Generate a new random salt.  
 /// </summary>  
 string CreateSaltKey(int size = 32);  
  
 /// <summary>  
 /// Verify a password against hash+salt.  
 /// </summary>  
 bool VerifyPassword(string enteredPassword, string storedHash, string storedSalt, PasswordFormat format);  
  
 /// <summary>  
 /// Generate a random secure token (for password reset, MFA).  
 /// </summary>  
 string GenerateSecureToken(int size = 64);  
  
 /// <summary>  
 /// Encrypt plain text with system key.  
 /// </summary>  
 string EncryptText(string plainText);  
  
 /// <summary>  
 /// Decrypt text with system key.  
 /// </summary>  
 string DecryptText(string cipherText);  
  
 /// <summary>  
 /// Compute SHA256 hash of a string.  
 /// </summary>  
 string ComputeSha256(string input);  
  
 /// <summary>  
 /// Compute HMAC (keyed hash).  
 /// </summary>  
 string ComputeHmac(string input, string key);  
 }  
}

## IPermissionService.cs

﻿using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Security;  
using App.Services.Common;  
  
namespace App.Services.Security  
{  
 public interface IPermissionService  
 {  
 // CRUD  
 Task<Permission> GetBySystemNameAsync(string systemName);  
 Task<IList<Permission>> GetAllAsync(bool onlyActive = true);  
 Task<ServiceResult<Permission>> InsertAsync(Permission permission);  
 Task<ServiceResult<Permission>> UpdateAsync(Permission permission);  
 Task<ServiceResult> DeleteAsync(int permissionId);  
  
 // Role ↔ Permission  
 Task<IList<string>> GetPermissionsForRoleAsync(int roleId);  
 Task<ServiceResult> GrantPermissionToRoleAsync(int roleId, string permissionSystemName);  
 Task<ServiceResult> RevokePermissionFromRoleAsync(int roleId, string permissionSystemName);  
  
 // User overrides  
 Task<IList<string>> GetUserOverridesAsync(int userId, bool onlyGranted = false);  
 Task<ServiceResult> SetUserOverrideAsync(int userId, string permissionSystemName, bool isGranted);  
 Task<ServiceResult> RemoveUserOverrideAsync(int userId, string permissionSystemName);  
  
 // Authorization check  
 Task<bool> AuthorizeAsync(int userId, string permissionSystemName);  
 }  
}

## IRoleService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Security;  
using App.Services.Common;  
  
namespace App.Services.Security  
{  
 public interface IRoleService  
 {  
 // CRUD  
 Task<Role> GetByIdAsync(int roleId);  
 Task<IList<Role>> GetAllAsync(bool onlyActive = true);  
 Task<ServiceResult<Role>> InsertAsync(Role role);  
 Task<ServiceResult<Role>> UpdateAsync(Role role);  
 Task<ServiceResult> DeleteAsync(int roleId);  
  
 Task<IList<Role>> GetRolesByUserIdAsync(int userId);  
 Task<ServiceResult> AddUserToRoleAsync(int userId, int roleId);  
 Task<ServiceResult> RemoveUserFromRoleAsync(int userId, int roleId);  
 }  
}

## PermissionService.cs

using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Security  
{  
 public class PermissionService : BaseService, IPermissionService  
 {  
 private readonly IRepository<Permission> \_permissionRepo;  
 private readonly IRepository<RolePermission> \_rolePermissionRepo;  
 private readonly IRepository<UserRole> \_userRoleRepo;  
 private readonly IRepository<UserPermissionOverride> \_userOverrideRepo;  
 private readonly ILocalizationService \_localization;  
  
 public PermissionService(  
 IRepository<Permission> permissionRepo,  
 IRepository<RolePermission> rolePermissionRepo,  
 IRepository<UserRole> userRoleRepo,  
 IRepository<UserPermissionOverride> userOverrideRepo,  
 ILocalizationService localization)  
 {  
 \_permissionRepo = permissionRepo;  
 \_rolePermissionRepo = rolePermissionRepo;  
 \_userRoleRepo = userRoleRepo;  
 \_userOverrideRepo = userOverrideRepo;  
 \_localization = localization;  
 }  
  
 #region CRUD  
  
 public async Task<Permission> GetBySystemNameAsync(string systemName)  
 {  
 var list = await \_permissionRepo.GetAllAsync(q => q.Where(p => p.SystemName == systemName));  
 return list.FirstOrDefault();  
 }  
  
 public async Task<IList<Permission>> GetAllAsync(bool onlyActive = true)  
 {  
 return await \_permissionRepo.GetAllAsync(q =>  
 onlyActive ? q.Where(p => p.IsActive) : q);  
 }  
  
 public async Task<ServiceResult<Permission>> InsertAsync(Permission permission)  
 {  
 if (permission == null)  
 return Failed<Permission>(await \_localization.GetResourceAsync("Permission.Insert.Null"));  
  
 var existing = await GetBySystemNameAsync(permission.SystemName);  
 if (existing != null)  
 return Failed<Permission>(await \_localization.GetResourceAsync("Permission.Insert.Duplicate"));  
  
 await \_permissionRepo.InsertAsync(permission);  
 return Success(permission, await \_localization.GetResourceAsync("Permission.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<Permission>> UpdateAsync(Permission permission)  
 {  
 if (permission == null)  
 return Failed<Permission>(await \_localization.GetResourceAsync("Permission.Update.Null"));  
  
 await \_permissionRepo.UpdateAsync(permission);  
 return Success(permission, await \_localization.GetResourceAsync("Permission.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int permissionId)  
 {  
 var entity = await \_permissionRepo.GetByIdAsync(permissionId);  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotFound"));  
  
 await \_permissionRepo.DeleteAsync(entity);  
 return Success(await \_localization.GetResourceAsync("Permission.Delete.Success"));  
 }  
  
 #endregion  
  
 #region Role ? Permission  
  
 public async Task<IList<string>> GetPermissionsForRoleAsync(int roleId)  
 {  
 var perms = from rp in \_rolePermissionRepo.Table  
 join p in \_permissionRepo.Table on rp.PermissionId equals p.Id  
 where rp.RoleId == roleId && p.IsActive  
 select p.SystemName;  
  
 return perms.ToList();  
 }  
  
 public async Task<ServiceResult> GrantPermissionToRoleAsync(int roleId, string permissionSystemName)  
 {  
 var perm = await GetBySystemNameAsync(permissionSystemName);  
 if (perm == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotFound"));  
  
 var exists = \_rolePermissionRepo.Table.Any(rp => rp.RoleId == roleId && rp.PermissionId == perm.Id);  
 if (exists)  
 return Failed(await \_localization.GetResourceAsync("Permission.AlreadyGranted"));  
  
 await \_rolePermissionRepo.InsertAsync(new RolePermission  
 {  
 RoleId = roleId,  
 PermissionId = perm.Id  
 });  
  
 return Success(await \_localization.GetResourceAsync("Permission.Grant.Success"));  
 }  
  
 public async Task<ServiceResult> RevokePermissionFromRoleAsync(int roleId, string permissionSystemName)  
 {  
 var perm = await GetBySystemNameAsync(permissionSystemName);  
 if (perm == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotFound"));  
  
 var entity = \_rolePermissionRepo.Table  
 .FirstOrDefault(rp => rp.RoleId == roleId && rp.PermissionId == perm.Id);  
  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotGranted"));  
  
 await \_rolePermissionRepo.DeleteAsync(entity);  
 return Success(await \_localization.GetResourceAsync("Permission.Revoke.Success"));  
 }  
  
 #endregion  
  
 #region User overrides  
  
 public async Task<IList<string>> GetUserOverridesAsync(int userId, bool onlyGranted = false)  
 {  
 var query = from o in \_userOverrideRepo.Table  
 join p in \_permissionRepo.Table on o.PermissionId equals p.Id  
 where o.UserId == userId  
 select new { o.IsGranted, p.SystemName };  
  
 if (onlyGranted)  
 return query.Where(x => x.IsGranted).Select(x => x.SystemName).ToList();  
  
 return query.Select(x => $"{x.SystemName}:{(x.IsGranted ? "Allow" : "Deny")}").ToList();  
 }  
  
 public async Task<ServiceResult> SetUserOverrideAsync(int userId, string permissionSystemName, bool isGranted)  
 {  
 var perm = await GetBySystemNameAsync(permissionSystemName);  
 if (perm == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotFound"));  
  
 var entity = \_userOverrideRepo.Table  
 .FirstOrDefault(o => o.UserId == userId && o.PermissionId == perm.Id);  
  
 if (entity != null)  
 {  
 entity.IsGranted = isGranted;  
 await \_userOverrideRepo.UpdateAsync(entity);  
 }  
 else  
 {  
 await \_userOverrideRepo.InsertAsync(new UserPermissionOverride  
 {  
 UserId = userId,  
 PermissionId = perm.Id,  
 IsGranted = isGranted  
 });  
 }  
  
 return Success(await \_localization.GetResourceAsync("Permission.Override.Success"));  
 }  
  
 public async Task<ServiceResult> RemoveUserOverrideAsync(int userId, string permissionSystemName)  
 {  
 var perm = await GetBySystemNameAsync(permissionSystemName);  
 if (perm == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.NotFound"));  
  
 var entity = \_userOverrideRepo.Table  
 .FirstOrDefault(o => o.UserId == userId && o.PermissionId == perm.Id);  
  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("Permission.Override.NotFound"));  
  
 await \_userOverrideRepo.DeleteAsync(entity);  
 return Success(await \_localization.GetResourceAsync("Permission.Override.Removed"));  
 }  
  
 #endregion  
  
 #region Authorization  
  
 public async Task<bool> AuthorizeAsync(int userId, string permissionSystemName)  
 {  
 var perm = await GetBySystemNameAsync(permissionSystemName);  
 if (perm == null || !perm.IsActive)  
 return false;  
  
 // 1) User override  
 var overrideEntity = \_userOverrideRepo.Table  
 .FirstOrDefault(o => o.UserId == userId && o.PermissionId == perm.Id);  
 if (overrideEntity != null)  
 return overrideEntity.IsGranted;  
  
 // 2) Roles  
 var roleIds = \_userRoleRepo.Table  
 .Where(ur => ur.UserId == userId)  
 .Select(ur => ur.RoleId)  
 .ToList();  
  
 if (!roleIds.Any())  
 return false;  
  
 var granted = \_rolePermissionRepo.Table  
 .Any(rp => roleIds.Contains(rp.RoleId) && rp.PermissionId == perm.Id);  
  
 return granted;  
 }  
  
 #endregion  
 }  
}

## RoleService.cs

using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Security  
{  
 public class RoleService : BaseService, IRoleService  
 {  
 private readonly IRepository<Role> \_roleRepo;  
 private readonly IRepository<UserRole> \_userRoleRepo;  
 private readonly ILocalizationService \_localization;  
  
 public RoleService(  
 IRepository<Role> roleRepo,  
 IRepository<UserRole> userRoleRepo,  
 ILocalizationService localization)  
 {  
 \_roleRepo = roleRepo;  
 \_userRoleRepo = userRoleRepo;  
 \_localization = localization;  
 }  
  
 #region CRUD  
  
 public async Task<Role> GetByIdAsync(int roleId)  
 => await \_roleRepo.GetByIdAsync(roleId);  
  
 public async Task<IList<Role>> GetAllAsync(bool onlyActive = true)  
 {  
 return await \_roleRepo.GetAllAsync(q =>  
 onlyActive ? q.Where(r => r.IsActive) : q);  
 }  
  
 public async Task<ServiceResult<Role>> InsertAsync(Role role)  
 {  
 if (role == null)  
 return Failed<Role>(await \_localization.GetResourceAsync("Role.Insert.Null"));  
  
 // SystemName unique check  
 var exists = \_roleRepo.Table.Any(r => r.SystemName == role.SystemName);  
 if (exists)  
 return Failed<Role>(await \_localization.GetResourceAsync("Role.Insert.Duplicate"));  
  
 role.CreatedOnUtc = System.DateTime.UtcNow;  
 await \_roleRepo.InsertAsync(role);  
  
 return Success(role, await \_localization.GetResourceAsync("Role.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<Role>> UpdateAsync(Role role)  
 {  
 if (role == null)  
 return Failed<Role>(await \_localization.GetResourceAsync("Role.Update.Null"));  
  
 role.UpdatedOnUtc = System.DateTime.UtcNow;  
 await \_roleRepo.UpdateAsync(role);  
  
 return Success(role, await \_localization.GetResourceAsync("Role.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int roleId)  
 {  
 var entity = await \_roleRepo.GetByIdAsync(roleId);  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("Role.NotFound"));  
  
 await \_roleRepo.DeleteAsync(entity);  
 return Success(await \_localization.GetResourceAsync("Role.Delete.Success"));  
 }  
  
 #endregion  
  
 #region User ? Role  
  
 public async Task<IList<Role>> GetRolesByUserIdAsync(int userId)  
 {  
 var query = from ur in \_userRoleRepo.Table  
 join r in \_roleRepo.Table on ur.RoleId equals r.Id  
 where ur.UserId == userId && r.IsActive  
 select r;  
  
 return query.ToList();  
 }  
  
 public async Task<ServiceResult> AddUserToRoleAsync(int userId, int roleId)  
 {  
 var exists = \_userRoleRepo.Table.Any(ur => ur.UserId == userId && ur.RoleId == roleId);  
 if (exists)  
 return Failed(await \_localization.GetResourceAsync("Role.User.AlreadyAssigned"));  
  
 await \_userRoleRepo.InsertAsync(new UserRole  
 {  
 UserId = userId,  
 RoleId = roleId  
 });  
  
 return Success(await \_localization.GetResourceAsync("Role.User.Added"));  
 }  
  
 public async Task<ServiceResult> RemoveUserFromRoleAsync(int userId, int roleId)  
 {  
 var entity = \_userRoleRepo.Table  
 .FirstOrDefault(ur => ur.UserId == userId && ur.RoleId == roleId);  
  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("Role.User.NotAssigned"));  
  
 await \_userRoleRepo.DeleteAsync(entity);  
 return Success(await \_localization.GetResourceAsync("Role.User.Removed"));  
 }  
  
 #endregion  
 }  
}

## ISettingService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace App.Services.Settings  
{  
 public interface ISettingService  
 {  
 Task<string> GetAsync(string key, string defaultValue = null);  
 Task<T> GetAsync<T>(string key, T defaultValue = default);  
 Task SetAsync(string key, string value);  
 Task SetAsync<T>(string key, T value);  
 Task<IDictionary<string, string>> GetAllAsync(string prefix = null);  
 Task<bool> HasKeyAsync(string key);  
 }  
}

## SettingService.cs

using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Settings;  
using App.Core.RepositoryServices;  
  
namespace App.Services.Settings  
{  
 public class SettingService : ISettingService  
 {  
 private readonly IRepository<Setting> \_settingRepository;  
  
 public SettingService(IRepository<Setting> settingRepository)  
 {  
 \_settingRepository = settingRepository;  
 }  
  
 public async Task<string> GetAsync(string key, string defaultValue = null)  
 {  
 var s = (await \_settingRepository.GetAllAsync(q => q.Where(x => x.Name == key))).FirstOrDefault();  
 return s?.Value ?? defaultValue;  
 }  
  
 public async Task<T> GetAsync<T>(string key, T defaultValue = default)  
 {  
 var val = await GetAsync(key);  
 if (val == null)  
 return defaultValue;  
  
 try  
 {  
 return (T)System.Convert.ChangeType(val, typeof(T));  
 }  
 catch  
 {  
 return defaultValue;  
 }  
 }  
  
 public async Task SetAsync(string key, string value)  
 {  
 var existing = (await \_settingRepository.GetAllAsync(q => q.Where(x => x.Name == key))).FirstOrDefault();  
 if (existing == null)  
 await \_settingRepository.InsertAsync(new Setting { Name = key, Value = value });  
 else  
 {  
 existing.Value = value;  
 await \_settingRepository.UpdateAsync(existing);  
 }  
 }  
  
 public async Task SetAsync<T>(string key, T value)  
 {  
 await SetAsync(key, value?.ToString());  
 }  
  
 public async Task<IDictionary<string, string>> GetAllAsync(string prefix = null)  
 {  
 var all = await \_settingRepository.GetAllAsync(q =>  
 string.IsNullOrWhiteSpace(prefix) ? q : q.Where(x => x.Name.StartsWith(prefix)));  
 return all.ToDictionary(x => x.Name, x => x.Value);  
 }  
  
 public async Task<bool> HasKeyAsync(string key)  
 {  
 var any = await \_settingRepository.GetAllAsync(q => q.Where(x => x.Name == key).Take(1));  
 return any.Any();  
 }  
 }  
}

## AppServicesStartup.cs

// <auto-generated />  
using Microsoft.Extensions.DependencyInjection;  
using App.Services.Audit;  
using App.Services.Calendar;  
using App.Services.Common;  
using App.Services.Correspondences;  
using App.Services.Directory;  
using App.Services.Files;  
using App.Services.Installation;  
using App.Services.Institutions;  
using App.Services.Localization;  
using App.Services.Notifications;  
using App.Services.Registrations;  
using App.Services.Reports;  
using App.Services.Security;  
using App.Services.Settings;  
using App.Services.Templates;  
using App.Services.Users;  
  
namespace App.Services  
{  
 /// <summary>  
 /// Central DI registration for App.Services  
 /// </summary>  
 public static class AppServicesStartup  
 {  
 /// <summary>  
 /// Registers all service interfaces to their implementations.  
 /// Call this from Program.cs: builder.Services.AddAppServices();  
 /// </summary>  
 public static IServiceCollection AddAppServices(this IServiceCollection services)  
 {  
 return services;  
 }  
 }  
}

## AppStartup.cs

﻿using App.Core;  
using App.Core.AppSettingsConfig;  
using App.Core.Caching;  
using App.Core.Configuration;  
using App.Core.Domain.Users;  
using App.Core.EngineServices;  
using App.Core.Infrastructure;  
using App.Core.MigratorServices;  
using App.Core.RepositoryServices;  
using App.Core.Security;  
using App.Core.Singletons;  
using App.Core.StartupServices;  
using App.Data.Configuration;  
using App.Data.DataProviders;  
using App.Data.Migrations;  
using App.Data.MigratorServices;  
using App.Services.Audit;  
using App.Services.Authentication;  
using App.Services.Common;  
using App.Services.Directory;  
using App.Services.Installation;  
using App.Services.Institutions;  
using App.Services.Localization;  
using App.Services.Notifications;  
using App.Services.Registrations;  
using App.Services.Reports;  
using App.Services.Security;  
using App.Services.Settings;  
using App.Services.Templates;  
using App.Services.Users;  
using FluentMigrator;  
using FluentMigrator.Runner;  
using FluentMigrator.Runner.VersionTableInfo;  
using Microsoft.AspNetCore.Builder;  
using Microsoft.AspNetCore.Hosting;  
using Microsoft.AspNetCore.Mvc.Infrastructure;  
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Reflection;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace App.Services.Startup  
{  
 public partial class AppStartup : IAppStartup  
 {  
 public void ConfigureServices(IServiceCollection services, IConfiguration configuration)  
  
 {  
 var loaded = AppDomain.CurrentDomain.GetAssemblies()  
 .Select(a => a.GetName().Name).OrderBy(n => n);  
 Console.WriteLine("Loaded assemblies: " + string.Join(", ", loaded));  
  
 var isAppDataLoaded = AppDomain.CurrentDomain.GetAssemblies()  
 .Any(a => a.GetName().Name.Equals("App.Data", StringComparison.OrdinalIgnoreCase));  
 Console.WriteLine("App.Data loaded: " + isAppDataLoaded);  
  
 // Add services to the container  
 services.AddControllers();  
 services.AddRazorPages();  
 services.AddHttpContextAccessor();  
 services.AddAuthentication("ApplicationCookie")  
 .AddCookie("ApplicationCookie", options =>  
 {  
 options.LoginPath = "/User/Login";  
 options.LogoutPath = "/User/Logout";  
 });  
 services.AddLocalization(options => options.ResourcesPath = "Resources");  
  
 // Register AppSettings  
 services.AddSingleton<AppSettings>();  
  
 // Register IAppFileProvider  
 services.AddSingleton<IAppFileProvider, AppFileProvider>();  
  
 // Register IStaticCacheManager  
 services.AddSingleton<IStaticCacheManager, MemoryCacheManager>();  
 services.AddScoped<ISettingService, SettingService>();  
  
 // Register IInstallationService  
 services.AddScoped<IInstallationService, InstallationService>();  
 services.AddTransient<InstallRequiredData>();  
 services.AddTransient<InstallSampleData>();  
  
 // Register IAppDataProvider  
 services.AddScoped<IAppDataProvider, MsSqlNopDataProvider>();  
  
 // Register IRepository<T>  
 services.AddScoped(typeof(IRepository<>), typeof(EntityRepository<>));  
  
 // Register IUserService  
 services.AddScoped<IUserService, UserService>();  
 services.AddScoped<IRoleService, RoleService>();  
 services.AddScoped<IAuthenticationService, AuthenticationService>();  
 services.AddScoped<IUserSettingsService, UserSettingsService>();  
 services.AddScoped<IInstitutionService, InstitutionService>();  
 services.AddScoped<ICountryService, CountryService>();  
 services.AddScoped<IStateProvinceService, StateProvinceService>();  
 services.AddScoped<ITemplateService, TemplateService>();  
 services.AddScoped<IReportService, ReportService>();  
  
 services.AddSingleton<ICacheKeyManager, CacheKeyManager>();  
 services.AddSingleton<UserSettings>();  
 services.AddTransient(typeof(IConcurrentCollection<>), typeof(ConcurrentTrie<>));  
 services.AddSingleton(provider =>  
 configuration.GetSection("UserSettings").Get<UserSettings>() ?? new UserSettings());  
 // Register IWorkContext  
 services.AddScoped<IShortTermCacheManager, PerRequestCacheManager>();  
 services.AddScoped<IWorkContext, WorkContext>();  
 // Register ILocalizationService  
 services.AddScoped<ILocalizationService, LocalizationService>();  
  
 //IWorkContext workContext = null;  
 services.AddTransient<Lazy<IWorkContext>>();  
 services.AddScoped<IGenericAttributeService, GenericAttributeService>();  
 services.AddScoped<IEncryptionService, EncryptionService>();  
 services.AddSingleton<SecuritySettings>();  
 services.AddSingleton<IActionContextAccessor, ActionContextAccessor>();  
  
 // Register IAccessControlService  
 services.AddScoped<IAccessControlService, AccessControlService>();  
  
 // Register INotificationService  
 services.AddScoped<INotificationService, NotificationService>();  
  
 // Register IAuditTrailService  
 services.AddScoped<IAuditTrailService, AuditTrailService>();  
 services.AddScoped<IRegistrationService, RegistrationService>();  
 services.AddScoped<IRegistrationSearchService, RegistrationSearchService>();  
  
 // Ensure Singleton<IMigrationManager> is initialized after FluentMigrator is configured  
 }  
  
 public void Configure(IApplicationBuilder app)  
 {  
  
 }  
  
 public int Order => 100;  
 }  
}

## ITemplateService.cs

using System.Collections.Generic;  
using System.Threading.Tasks;  
using App.Core.Domain.Common;  
using App.Services.Common;  
  
namespace App.Services.Templates  
{  
 public interface ITemplateService  
 {  
 Task<ServiceResult<Template>> GetAsync(int id);  
 Task<ServiceResult<IReadOnlyList<Template>>> GetAllAsync(string type = null);  
 Task<ServiceResult<Template>> CreateAsync(Template model);  
 Task<ServiceResult> UpdateAsync(int id, Template model);  
 Task<ServiceResult> DeleteAsync(int id);  
 }  
}

## TemplateService.cs

using System.Linq;  
using System.Threading.Tasks;  
using System.Collections.Generic;  
using App.Core.Domain.Common;  
using App.Core.RepositoryServices;  
using App.Services.Common;  
using App.Services.Localization;  
  
namespace App.Services.Templates  
{  
 public class TemplateService : BaseService, ITemplateService  
 {  
 private readonly IRepository<Template> \_templateRepository;  
 private readonly ILocalizationService \_localizationService;  
  
 public TemplateService(IRepository<Template> templateRepository, ILocalizationService localizationService)  
 {  
 \_templateRepository = templateRepository;  
 \_localizationService = localizationService;  
 }  
  
 public async Task<ServiceResult<Template>> GetAsync(int id)  
 {  
 var entity = await \_templateRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult<Template>.Failed(await \_localizationService.GetResourceAsync("Template.NotFound"));  
 return ServiceResult<Template>.Success(entity);  
 }  
  
 public async Task<ServiceResult<IReadOnlyList<Template>>> GetAllAsync(string type = null)  
 {  
 var list = await \_templateRepository.GetAllAsync(q =>  
 {  
 if (!string.IsNullOrWhiteSpace(type)) q = q.Where(x => x.TemplateType == type);  
 return q;  
 });  
 return ServiceResult<IReadOnlyList<Template>>.Success(list.ToList());  
 }  
  
 public async Task<ServiceResult<Template>> CreateAsync(Template model)  
 {  
 if (model == null) return ServiceResult<Template>.Failed(await \_localizationService.GetResourceAsync("Errors.NullModel"));  
 await \_templateRepository.InsertAsync(model);  
 return ServiceResult<Template>.Success(model);  
 }  
  
 public async Task<ServiceResult> UpdateAsync(int id, Template model)  
 {  
 var entity = await \_templateRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Template.NotFound"));  
 entity.Name = model.Name;  
 entity.TemplateType = model.TemplateType;  
 entity.Content = model.Content;  
 await \_templateRepository.UpdateAsync(entity);  
 return ServiceResult.SuccessResult();  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var entity = await \_templateRepository.GetByIdAsync(id);  
 if (entity == null) return ServiceResult.Failed(await \_localizationService.GetResourceAsync("Template.NotFound"));  
 await \_templateRepository.DeleteAsync(entity);  
 return ServiceResult.SuccessResult();  
 }  
 }  
}

## IUserDirectory.cs

using System.Threading.Tasks;  
  
namespace App.Services.Users  
{  
 /// <summary>  
 /// Facade used by Admin/UserManagementController to manage user lifecycle and role assignments.  
 /// Bridges to IUserService for activate/deactivate and handles role mapping.  
 /// </summary>  
 public interface IUserDirectory  
 {  
 Task<bool> ActivateAsync(int userId);  
 Task<bool> DeactivateAsync(int userId);  
 /// <summary>  
 /// Assign roles to a user by role SystemName or Name. Replaces existing mappings.  
 /// </summary>  
 Task<bool> AssignRolesAsync(int userId, string[] roles);  
 }  
}

## IUserService.cs

using App.Core.Domain.Audit;  
using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.Domain.Users.App.Core.Domain.Users;  
using App.Services.Common;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace App.Services.Users  
{  
 public interface IUserService  
 {  
 // CRUD  
 Task<User> GetByIdAsync(int id);  
 Task<IList<User>> GetAllAsync(bool onlyActive = true);  
 Task<ServiceResult<User>> InsertAsync(User user, string password, int createdByUserId);  
 Task<ServiceResult<User>> UpdateAsync(User user);  
 Task<ServiceResult> DeleteAsync(int id);  
  
 // Identity  
 Task<User> GetByEmailAsync(string email);  
 Task<User> GetByUsernameAsync(string username);  
  
 // Security  
 Task<ServiceResult> ActivateAsync(int userId);  
 Task<ServiceResult> DeactivateAsync(int userId);  
 Task<ServiceResult> ResetPasswordAsync(int userId, string newPassword, int performedByUserId);  
 Task<ServiceResult> LockUserAsync(int userId, int? minutes = null);  
 Task<ServiceResult> UnlockUserAsync(int userId);  
 Task<bool> CheckPasswordAsync(User user, string password);  
 Task<bool> IsPasswordExpiredAsync(User user);  
  
 // Roles  
 Task<IList<string>> GetRolesAsync(int userId);  
 Task<IList<Role>> GetRoleEntitiesAsync(int userId);  
 Task<bool> IsInRoleAsync(int userId, string roleSystemName);  
 Task<ServiceResult> AddToRoleAsync(int userId, int roleId);  
 Task<ServiceResult> RemoveFromRoleAsync(int userId, int roleId);  
  
 // Status Queries  
 Task<IList<User>> GetLockedUsersAsync();  
 Task<IList<User>> GetInactiveUsersAsync();  
  
 // Preferences  
 Task<UserPreference> GetPreferencesAsync(int userId);  
 Task<ServiceResult> UpdatePreferencesAsync(UserPreference preference);  
  
 // Audit  
 Task<IList<AuditTrail>> GetUserAuditTrailAsync(int userId, int pageIndex = 0, int pageSize = 50);  
 }  
  
}

## IUserSettingsService.cs

using App.Core.Domain.Users;  
using App.Core.Domain.Users.App.Core.Domain.Users;  
using System.Threading.Tasks;  
  
namespace App.Services.Users  
{  
 public interface IUserSettingsService  
 {  
 Task<UserPreference> GetByUserIdAsync(int userId);  
  
 Task SetLanguageAsync(int userId, int languageId);  
  
 Task ToggleMfaAsync(int userId, bool enable);  
  
 Task UpdateNotificationPreferencesAsync(  
 int userId,  
 bool notifyByEmail,  
 bool notifyBySms,  
 bool notifyInApp);  
 }  
}

## UserDirectory.cs

using System;  
using System.Linq;  
using System.Threading.Tasks;  
using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.RepositoryServices;  
  
namespace App.Services.Users  
{  
 public class UserDirectory : IUserDirectory  
 {  
 private readonly IUserService \_userService;  
 private readonly IRepository<Role> \_roleRepository;  
 private readonly IRepository<UserRole> \_userRoleRepository;  
 private readonly IRepository<User> \_userRepository;  
  
 public UserDirectory(  
 IUserService userService,  
 IRepository<Role> roleRepository,  
 IRepository<UserRole> userRoleRepository,  
 IRepository<User> userRepository)  
 {  
 \_userService = userService;  
 \_roleRepository = roleRepository;  
 \_userRoleRepository = userRoleRepository;  
 \_userRepository = userRepository;  
 }  
  
 public async Task<bool> ActivateAsync(int userId)  
 {  
 var res = await \_userService.ActivateAsync(userId);  
 return res.Success;  
 }  
  
 public async Task<bool> DeactivateAsync(int userId)  
 {  
 var res = await \_userService.DeactivateAsync(userId);  
 return res.Success;  
 }  
  
 public async Task<bool> AssignRolesAsync(int userId, string[] roles)  
 {  
 if (roles == null) return false;  
  
 var user = await \_userRepository.GetByIdAsync(userId);  
 if (user == null) return false;  
  
 // Fix for CS0121: Specify default parameters to resolve ambiguity  
 var allRoles = await \_roleRepository.GetAllAsync((IQueryable<Role> q) => q, true);  
 var roleIds = allRoles  
 .Where(r => roles.Any(x => string.Equals(x, r.SystemName, StringComparison.OrdinalIgnoreCase)  
 || string.Equals(x, r.Name, StringComparison.OrdinalIgnoreCase)))  
 .Select(r => r.Id)  
 .Distinct()  
 .ToList();  
  
 // Remove current mappings  
 var existing = await \_userRoleRepository.GetAllAsync(q => q.Where(ur => ur.UserId == userId));  
 if (existing.Any())  
 await \_userRoleRepository.DeleteAsync(existing);  
  
 // Add new mappings  
 foreach (var rid in roleIds)  
 await \_userRoleRepository.InsertAsync(new UserRole { UserId = userId, RoleId = rid });  
  
 return true;  
 }  
 }  
}

## UserService.cs

using App.Core.Configuration;  
using App.Core.Domain.Audit;  
using App.Core.Domain.Security;  
using App.Core.Domain.Users;  
using App.Core.Domain.Users.App.Core.Domain.Users;  
using App.Core.RepositoryServices;  
using App.Services.Audit;  
using App.Services.Common;  
using App.Services.Localization;  
using App.Services.Security;  
using Microsoft.EntityFrameworkCore;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace App.Services.Users  
{  
 public class UserService : BaseService, IUserService  
 {  
 private readonly IRepository<User> \_userRepo;  
 private readonly IRepository<UserRole> \_userRoleRepo;  
 private readonly IRepository<Role> \_roleRepo;  
 private readonly IRepository<UserPreference> \_preferenceRepo;  
 private readonly IEncryptionService \_encryptionService;  
 private readonly ILocalizationService \_localization;  
 private readonly SecuritySettings \_securitySettings;  
 private readonly IAuditTrailService \_audit;  
  
 public UserService(  
 IRepository<User> userRepo,  
 IRepository<UserRole> userRoleRepo,  
 IRepository<Role> roleRepo,  
 IRepository<UserPreference> preferenceRepo,  
 IEncryptionService encryptionService,  
 ILocalizationService localization,  
 SecuritySettings securitySettings,  
 IAuditTrailService audit)  
 {  
 \_userRepo = userRepo;  
 \_userRoleRepo = userRoleRepo;  
 \_roleRepo = roleRepo;  
 \_preferenceRepo = preferenceRepo;  
 \_encryptionService = encryptionService;  
 \_localization = localization;  
 \_securitySettings = securitySettings;  
 \_audit = audit;  
 }  
  
 #region CRUD  
  
 public async Task<User> GetByIdAsync(int id) =>  
 await \_userRepo.GetByIdAsync(id);  
  
 public async Task<IList<User>> GetAllAsync(bool onlyActive = true)  
 {  
 var query = \_userRepo.Table.AsQueryable();  
 if (onlyActive) query = query.Where(u => u.IsActive);  
 return await query.ToListAsync();  
 }  
  
 public async Task<ServiceResult<User>> InsertAsync(User user, string password, int createdByUserId)  
 {  
 if (string.IsNullOrWhiteSpace(password))  
 return Failed<User>(await \_localization.GetResourceAsync("User.Password.Required"));  
  
 var salt = \_encryptionService.CreateSaltKey(\_securitySettings.PasswordSaltSize);  
 var hash = \_encryptionService.CreatePasswordHash(password, salt, PasswordFormat.Hashed);  
  
 user.PasswordSalt = salt;  
 user.PasswordHash = hash;  
 user.CreatedOnUtc = DateTime.UtcNow;  
 user.IsActive = true;  
  
 await \_userRepo.InsertAsync(user);  
  
 await \_audit.LogCreateAsync(nameof(User), user.Id, createdByUserId, "User created");  
  
 return Success(user, await \_localization.GetResourceAsync("User.Insert.Success"));  
 }  
  
 public async Task<ServiceResult<User>> UpdateAsync(User user)  
 {  
 var before = await \_userRepo.GetByIdAsync(user.Id);  
 if (before == null)  
 return Failed<User>(await \_localization.GetResourceAsync("User.NotFound"));  
  
 user.UpdatedOnUtc = DateTime.UtcNow;  
  
 await \_userRepo.UpdateAsync(user);  
  
 await AuditDiffAsync(before, user, 0, "User updated");  
  
 return Success(user, await \_localization.GetResourceAsync("User.Update.Success"));  
 }  
  
 public async Task<ServiceResult> DeleteAsync(int id)  
 {  
 var entity = await \_userRepo.GetByIdAsync(id);  
 if (entity == null)  
 return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 await \_userRepo.DeleteAsync(entity);  
  
 await \_audit.LogDeleteAsync(nameof(User), id, 0, "User deleted");  
  
 return Success(await \_localization.GetResourceAsync("User.Delete.Success"));  
 }  
  
 #endregion  
  
 #region Identity  
  
 public async Task<User> GetByEmailAsync(string email) =>  
 await \_userRepo.Table.FirstOrDefaultAsync(u => u.Email == email);  
  
 public async Task<User> GetByUsernameAsync(string username) =>  
 await \_userRepo.Table.FirstOrDefaultAsync(u => u.Username == username);  
  
 #endregion  
  
 #region Security  
  
 public async Task<ServiceResult> ActivateAsync(int userId)  
 {  
 var user = await \_userRepo.GetByIdAsync(userId);  
 if (user == null) return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 var old = user.IsActive;  
 user.IsActive = true;  
 user.DeactivatedOnUtc = null;  
 await \_userRepo.UpdateAsync(user);  
  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, userId, "IsActive", old.ToString(), "True", "Activate");  
  
 return Success(await \_localization.GetResourceAsync("User.Activate.Success"));  
 }  
  
 public async Task<ServiceResult> DeactivateAsync(int userId)  
 {  
 var user = await \_userRepo.GetByIdAsync(userId);  
 if (user == null) return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 var old = user.IsActive;  
 user.IsActive = false;  
 user.DeactivatedOnUtc = DateTime.UtcNow;  
 await \_userRepo.UpdateAsync(user);  
  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, 0, "IsActive", old.ToString(), "False", "Deactivate");  
  
 return Success(await \_localization.GetResourceAsync("User.Deactivate.Success"));  
 }  
  
 public async Task<ServiceResult> ResetPasswordAsync(int userId, string newPassword, int performedByUserId)  
 {  
 var user = await \_userRepo.GetByIdAsync(userId);  
 if (user == null) return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 var salt = \_encryptionService.CreateSaltKey(\_securitySettings.PasswordSaltSize);  
 var hash = \_encryptionService.CreatePasswordHash(newPassword, salt, PasswordFormat.Hashed);  
  
 user.PasswordSalt = salt;  
 user.PasswordHash = hash;  
 user.UpdatedOnUtc = DateTime.UtcNow;  
  
 await \_userRepo.UpdateAsync(user);  
  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, performedByUserId, "Password", null, null, "Password reset");  
  
 return Success(await \_localization.GetResourceAsync("User.ResetPassword.Success"));  
 }  
  
 public async Task<ServiceResult> LockUserAsync(int userId, int? minutes = null)  
 {  
 var user = await \_userRepo.GetByIdAsync(userId);  
 if (user == null) return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 var oldLocked = user.IsLockedOut;  
 var oldEnd = user.LockoutEndUtc?.ToString("o");  
  
 user.IsLockedOut = true;  
 user.LockoutEndUtc = minutes.HasValue  
 ? DateTime.UtcNow.AddMinutes(minutes.Value)  
 : DateTime.UtcNow.AddMinutes(\_securitySettings.DefaultLockoutMinutes);  
  
 await \_userRepo.UpdateAsync(user);  
  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, 0, "IsLockedOut", oldLocked.ToString(), "True", "Lock");  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, 0, "LockoutEndUtc", oldEnd, user.LockoutEndUtc?.ToString("o"));  
  
 return Success(await \_localization.GetResourceAsync("User.Lock.Success"));  
 }  
  
 public async Task<ServiceResult> UnlockUserAsync(int userId)  
 {  
 var user = await \_userRepo.GetByIdAsync(userId);  
 if (user == null) return Failed(await \_localization.GetResourceAsync("User.NotFound"));  
  
 var oldLocked = user.IsLockedOut;  
 var oldEnd = user.LockoutEndUtc?.ToString("o");  
  
 user.IsLockedOut = false;  
 user.LockoutEndUtc = null;  
  
 await \_userRepo.UpdateAsync(user);  
  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, 0, "IsLockedOut", oldLocked.ToString(), "False", "Unlock");  
 await \_audit.LogUpdateAsync(nameof(User), user.Id, 0, "LockoutEndUtc", oldEnd, null);  
  
 return Success(await \_localization.GetResourceAsync("User.Unlock.Success"));  
 }  
  
 public async Task<bool> CheckPasswordAsync(User user, string password)  
 {  
 var hash = \_encryptionService.CreatePasswordHash(password, user.PasswordSalt, PasswordFormat.Hashed);  
 return hash == user.PasswordHash;  
 }  
  
 public Task<bool> IsPasswordExpiredAsync(User user)  
 {  
 if (\_securitySettings.PasswordExpiryDays <= 0 || !user.UpdatedOnUtc.HasValue)  
 return Task.FromResult(false);  
  
 return Task.FromResult(user.UpdatedOnUtc.Value.AddDays(\_securitySettings.PasswordExpiryDays) < DateTime.UtcNow);  
 }  
  
 public async Task<IList<string>> GetRolesAsync(int userId)  
 {  
 // Fix: Join UserRole with Role using RoleId, then select Role.SystemName  
 var query = from ur in \_userRoleRepo.Table  
 join r in \_roleRepo.Table on ur.RoleId equals r.Id  
 where ur.UserId == userId  
 select r.SystemName;  
 return await query.ToListAsync();  
 }  
  
 public async Task<IList<Role>> GetRoleEntitiesAsync(int userId)  
 {  
 // Fix: Join UserRole with Role using RoleId, then select Role entity  
 var query = from ur in \_userRoleRepo.Table  
 join r in \_roleRepo.Table on ur.RoleId equals r.Id  
 where ur.UserId == userId  
 select r;  
 return await query.ToListAsync();  
 }  
  
 public async Task<bool> IsInRoleAsync(int userId, string roleSystemName)  
 {  
 // Fix: Join UserRole with Role using RoleId, then check Role.SystemName  
 var query = from ur in \_userRoleRepo.Table  
 join r in \_roleRepo.Table on ur.RoleId equals r.Id  
 where ur.UserId == userId && r.SystemName == roleSystemName  
 select ur;  
 var result = await query.ToListAsync();  
 return result.Any();  
 }  
  
 public async Task<ServiceResult> AddToRoleAsync(int userId, int roleId)  
 {  
 var exists = \_userRoleRepo.Table.Any(ur => ur.UserId == userId && ur.RoleId == roleId);  
 if (exists) return Failed(await \_localization.GetResourceAsync("User.Role.AlreadyAssigned"));  
  
 var before = await GetRolesAsync(userId);  
  
 await \_userRoleRepo.InsertAsync(new UserRole { UserId = userId, RoleId = roleId });  
  
 var after = await GetRolesAsync(userId);  
 var roleName = \_roleRepo.Table.Where(r => r.Id == roleId).Select(r => r.SystemName).FirstOrDefault();  
  
 await \_audit.LogUpdateAsync(nameof(User), userId, 0, "Roles",  
 string.Join(",", before), string.Join(",", after),  
 comment: $"Add role: {roleName}");  
  
 return Success(await \_localization.GetResourceAsync("User.Role.Assigned"));  
 }  
  
 public async Task<ServiceResult> RemoveFromRoleAsync(int userId, int roleId)  
 {  
 var entity = \_userRoleRepo.Table.FirstOrDefault(ur => ur.UserId == userId && ur.RoleId == roleId);  
 if (entity == null) return Failed(await \_localization.GetResourceAsync("User.Role.NotAssigned"));  
  
 var before = await GetRolesAsync(userId);  
  
 await \_userRoleRepo.DeleteAsync(entity);  
  
 var after = await GetRolesAsync(userId);  
 var roleName = \_roleRepo.Table.Where(r => r.Id == roleId).Select(r => r.SystemName).FirstOrDefault();  
  
 await \_audit.LogUpdateAsync(nameof(User), userId, 0, "Roles",  
 string.Join(",", before), string.Join(",", after),  
 comment: $"Remove role: {roleName}");  
  
 return Success(await \_localization.GetResourceAsync("User.Role.Removed"));  
 }  
  
 #endregion  
  
 #region Status Queries  
  
 public async Task<IList<User>> GetLockedUsersAsync()  
 {  
 return await \_userRepo.Table  
 .Where(u => u.IsLockedOut)  
 .ToListAsync();  
 }  
  
 public async Task<IList<User>> GetInactiveUsersAsync()  
 {  
 return await \_userRepo.Table  
 .Where(u => !u.IsActive)  
 .ToListAsync();  
 }  
  
 #endregion  
  
 #region Preferences  
  
 public async Task<UserPreference> GetPreferencesAsync(int userId)  
 {  
 return await \_preferenceRepo.Table  
 .FirstOrDefaultAsync(p => p.UserId == userId);  
 }  
  
 public async Task<ServiceResult> UpdatePreferencesAsync(UserPreference preference)  
 {  
 var existing = await \_preferenceRepo.Table.FirstOrDefaultAsync(p => p.UserId == preference.UserId);  
 if (existing == null)  
 await \_preferenceRepo.InsertAsync(preference);  
 else  
 {  
 existing.LanguageId = preference.LanguageId;  
 existing.EnableMfa = preference.EnableMfa;  
 existing.NotifyByEmail = preference.NotifyByEmail;  
 existing.NotifyBySms = preference.NotifyBySms;  
 existing.NotifyInApp = preference.NotifyInApp;  
 existing.UpdatedOnUtc = DateTime.UtcNow;  
 await \_preferenceRepo.UpdateAsync(existing);  
 }  
  
 return Success(await \_localization.GetResourceAsync("User.Preference.Updated"));  
 }  
  
 #endregion  
  
 #region Audit  
  
 public async Task<IList<AuditTrail>> GetUserAuditTrailAsync(int userId, int pageIndex = 0, int pageSize = 50)  
 {  
 var result = await \_audit.SearchAsync(  
 changedByUserId: userId,  
 pageIndex: pageIndex,  
 pageSize: pageSize);  
  
 return result.Items;  
 }  
  
 #endregion  
  
 #region AuditDiff Helper  
  
 private async Task AuditDiffAsync(User before, User after, int changedByUserId, string comment = null)  
 {  
 if (before == null || after == null || before.Id != after.Id) return;  
  
 var props = typeof(User).GetProperties()  
 .Where(p => p.CanRead && p.GetIndexParameters().Length == 0 &&  
 p.Name != nameof(User.PasswordHash) &&  
 p.Name != nameof(User.PasswordSalt));  
  
 foreach (var p in props)  
 {  
 var oldVal = p.GetValue(before)?.ToString();  
 var newVal = p.GetValue(after)?.ToString();  
 if (!string.Equals(oldVal, newVal, StringComparison.Ordinal))  
 {  
 await \_audit.LogUpdateAsync(  
 nameof(User),  
 after.Id,  
 changedByUserId,  
 p.Name,  
 oldVal,  
 newVal,  
 comment);  
 }  
 }  
 }  
  
 #endregion  
 }  
}

## UserSettingsService.cs

using App.Core.Domain.Users;  
using App.Core.Domain.Users.App.Core.Domain.Users;  
using App.Core.RepositoryServices;  
using Microsoft.EntityFrameworkCore;  
using System;  
using System.Threading.Tasks;  
  
namespace App.Services.Users  
{  
 public class UserSettingsService : IUserSettingsService  
 {  
 private readonly IRepository<UserPreference> \_preferenceRepo;  
 private readonly ILanguageService \_languageService;  
  
 public UserSettingsService(  
 IRepository<UserPreference> preferenceRepo,  
 ILanguageService languageService)  
 {  
 \_preferenceRepo = preferenceRepo;  
 \_languageService = languageService;  
 }  
  
 public async Task<UserPreference> GetByUserIdAsync(int userId)  
 {  
 return await \_preferenceRepo.Table  
 .FirstOrDefaultAsync(p => p.UserId == userId);  
 }  
  
 public async Task SetLanguageAsync(int userId, int languageId)  
 {  
 // Ensure language exists  
 var lang = await \_languageService.GetByIdAsync(languageId);  
 if (lang == null)  
 throw new ArgumentException("Invalid languageId");  
  
 var pref = await EnsurePreferenceAsync(userId);  
 pref.LanguageId = languageId;  
 pref.UpdatedOnUtc = DateTime.UtcNow;  
 await \_preferenceRepo.UpdateAsync(pref);  
 }  
  
 public async Task ToggleMfaAsync(int userId, bool enable)  
 {  
 var pref = await EnsurePreferenceAsync(userId);  
 pref.EnableMfa = enable;  
 pref.UpdatedOnUtc = DateTime.UtcNow;  
 await \_preferenceRepo.UpdateAsync(pref);  
 }  
  
 public async Task UpdateNotificationPreferencesAsync(  
 int userId,  
 bool notifyByEmail,  
 bool notifyBySms,  
 bool notifyInApp)  
 {  
 var pref = await EnsurePreferenceAsync(userId);  
 pref.NotifyByEmail = notifyByEmail;  
 pref.NotifyBySms = notifyBySms;  
 pref.NotifyInApp = notifyInApp;  
 pref.UpdatedOnUtc = DateTime.UtcNow;  
 await \_preferenceRepo.UpdateAsync(pref);  
 }  
  
 private async Task<UserPreference> EnsurePreferenceAsync(int userId)  
 {  
 var pref = await \_preferenceRepo.Table.FirstOrDefaultAsync(p => p.UserId == userId);  
 if (pref == null)  
 {  
 pref = new UserPreference  
 {  
 UserId = userId,  
 LanguageId = null,  
 EnableMfa = false,  
 NotifyByEmail = true,  
 NotifyBySms = false,  
 NotifyInApp = true,  
 UpdatedOnUtc = DateTime.UtcNow  
 };  
 await \_preferenceRepo.InsertAsync(pref);  
 }  
 return pref;  
 }  
 }  
}

# ✅ Completed Services

- RegistrationActionHintsService (expanded, aligned with Domain)

- UserService (reimplemented correctly with User, UserPreference, AuditTrailService)

- PermissionService (validated, no issues)

- AccessControlService (validated, no issues)

- InstallationService + InstallRequiredData + InstallSampleData (rewritten to align with Domain entities)

- NotificationService.SendAsync integration with NotifyAsync fixed (customMessage handled via tokens)

# ⚠️ Services to Review

- RegistrationService (workflow methods, status transitions, notifications, status logs)

- AuditTrailService integration (ensure all Services log correctly: User, Registration, Contact, Document)

- ContactService (review structure and consistency)

- DocumentService (review structure and consistency)

- RegistrationStatusLogService (review structure and consistency)

# 📌 Instructions for GPT

عند قراءة هذا الدوكيومنت:  
1. اعتمد على كل ما هو مكتوب في Entities و Enums و Interfaces و Services في الأقسام الأولى.  
2. راجع القوائم في آخر الدوكيومنت:  
 - ✅ Completed Services → هذه خلصت ومش محتاجة أي تعديل.  
 - ⚠️ Services to Review → هذه محتاجة مراجعة أو إعادة كتابة.  
 - 🔧 Services to Expand → هذه موجودة بس محتاجة توسعة لتغطي كل السيناريوهات.  
3. المطلوب مع كل Service أو Interface:  
 • افتح الـ Entity أو الـ Enum المطلوب ووريني كل الـ Properties/Values اللي فيه.  
 • افتح الـ Interface المطلوب ووريني كل الـ Methods + Signatures بالظبط (اسم، براميترز، Return type).  
 • بعد كده ابني الكود (Service/Implementation) بحيث يعتمد فقط على اللي موجود فعلًا في الدوكيومنت.  
 • لو لقيت حاجة مش موجودة في الـ Domain أو الـ Interface، ما تفترضش إنها موجودة → قول إنها ناقصة ونضيفها كـ Feature.

# 🔧 Services to Expand

- RegistrationActionHintsService (needs more detailed validation rules, notifications mapping per role)

- ILookupService (currently enum-based, later expand to support dynamic DB-driven lookups)

# 📌 Services Status Reference (Merged Update)

# 📝 Supervision System – Services Status Reference   
  
هذا المرجع يوضح حالة جميع الـ Services الحالية في المشروع، وهو مخصص كـ Input لأي جلسة GPT جديدة.   
الهدف: استكمال ما لم يكتمل من Services، ثم الانتقال إلى طبقة الـ Web.   
  
---  
  
## ✅ Completed / أساسي شغال  
- \*\*UserService / IUserService\*\*   
 - CRUD أساسي جاهز.   
 - ينقصه فقط بعض التوسعة (Multi-role, Activation/Deactivation).   
  
- \*\*ContactService / IContactService\*\*   
 - CRUD موجود.   
 - محتاج مراجعة على Validation وربطه بالـ Registration.   
  
- \*\*DocumentService / IDocumentService\*\*   
 - رفع/تحميل ملفات شغال.   
 - ناقص Validation وMetadata.   
  
---  
  
## 🟡 Requires Review / محتاج مراجعة  
- \*\*RegistrationService / IRegistrationService\*\*   
 - Workflow ناقص (ReturnedForEdit, Archived).   
 - NotifyAsync لسه مثبت على RecipientId = 0.   
  
- \*\*NotificationService / INotificationService\*\*   
 - SendAsync موجود لكن محدود.   
 - ناقص دعم كامل للـ Channels (Email, SMS, In-App).   
  
- \*\*AuditTrailService / IAuditTrailService\*\*   
 - Logging أساسي موجود.   
 - مش بيغطي كل الـ Entities.   
  
---  
  
## ⚡ Requires Expansion / محتاج توسعة  
- \*\*Role & Permission Services (RBAC)\*\*   
 - Matrix مش مطبق بالكامل (Maker, Checker, Regulator, Admin, Inspector).   
 - ناقص Integration مع Workflow.   
  
- \*\*ReportingService / IReportingService\*\*   
 - Placeholder موجود.   
 - ناقص تقارير فعلية + Export CSV/PDF.   
  
---  
  
## ❌ Not Yet Implemented / لسه ناقص  
- \*\*Search & Filter APIs\*\*   
 - للـ Registration (بحث بالـ License, Country, Status).   
 - للـ AuditTrail (فلترة حسب User, Action, Date).   
  
- \*\*Advanced Notifications\*\*   
 - Workflow كامل (Submit, Approve, Reject, Return).   
 - Integration مع Email/SMS Provider.   
  
- \*\*Full Reporting Module\*\*   
 - حسب BRD (Regulator Reports, Monitoring, CSV/PDF Export).   
  
---  
  
## ➕ To Be Added / محتاجين نزوده  
- \*\*Integration Service\*\*: للتعامل مع External APIs (مستقبلاً).   
- \*\*Scheduler/Job Service\*\*: عشان إشعارات Expiry أو Deadlines.   
- \*\*Compliance Dashboard Service\*\*: عرض حالة الـ FI وStatuses لحظياً (BRD requirement).   
  
---  
  
## 🎯 Next Step  
- استكمال ما فوق (المراجعة + التوسعة + الإضافة).   
- بعد الانتهاء → \*\*نبدأ في طبقة الـ Web (Controllers + Views + API Integration)\*\*.