بر اساس اطلاعات موجود در فایل‌ها و محتوای ارائه‌شده، لیست کامل \*\*ماژول‌ها (Modules)\*\* به شرح زیر است:

1. \*\*ApiGateway\*\*

2. \*\*DeviceManagementService\*\*

3. \*\*LocationTrackingService\*\*

4. \*\*CommandService\*\*

5. \*\*NotificationService\*\*

6. \*\*AuthService\*\*

7. \*\*BillingService\*\*

8. \*\*SubscriptionService\*\*

9. \*\*SharedKernel\*\*

10. \*\*Infrastructure\*\*

11. \*\*EventBus\*\*

12. \*\*EventStoreService\*\*

13. \*\*JobSchedulerService\*\*

14. \*\*SystemMonitoringService\*\*

15. \*\*HealthCheckService\*\*

16. \*\*Tests\*\*

---

### \*\*توضیح کوتاه برای هر ماژول:\*\*

1. \*\*ApiGateway\*\*

- نقش دروازه ورودی سیستم و مدیریت توزیع درخواست‌ها به میکروسرویس‌های مختلف.

2. \*\*DeviceManagementService\*\*

- مدیریت دستگاه‌ها شامل ثبت، به‌روزرسانی، حذف، اعتبارسنجی شماره سریال، و اختصاص دستگاه‌ها به کاربران.

3. \*\*LocationTrackingService\*\*

- ردیابی موقعیت‌های دستگاه‌ها و مدیریت داده‌های مکانی.

4. \*\*CommandService\*\*

- مدیریت ارسال، ذخیره‌سازی، و پاسخ‌دهی به دستورات ارسالی به دستگاه‌ها.

5. \*\*NotificationService\*\*

- مدیریت اعلان‌ها و هشدارها برای کاربران از طریق SMS، ایمیل، و SignalR.

6. \*\*AuthService\*\*

- مدیریت احراز هویت، تولید JWT، پشتیبانی از MFA (Multi-Factor Authentication)، و ردیابی IP.

7. \*\*BillingService\*\*

- مدیریت صورتحساب‌ها، پرداخت‌ها، و وضعیت مالی کاربران.

8. \*\*SubscriptionService\*\*

- مدیریت اشتراک‌ها، تمدید اشتراک‌ها، و پرداخت‌های مرتبط.

9. \*\*SharedKernel\*\*

- شامل ابزارهای عمومی، Value Objects، Enums، Utilities، و قراردادهای مشترک بین ماژول‌ها.

10. \*\*Infrastructure\*\*

- مدیریت زیرساخت‌ها مانند دیتابیس، Redis، لاگ‌گیری، و Health Checks.

11. \*\*EventBus\*\*

- مدیریت رویدادها و انتشار آن‌ها بین میکروسرویس‌ها با استفاده از Kafka یا RabbitMQ.

12. \*\*EventStoreService\*\*

- ذخیره‌سازی رویدادها برای ردیابی تغییرات و بازیابی وضعیت سیستم.

13. \*\*JobSchedulerService\*\*

- مدیریت وظایف زمان‌بندی‌شده مانند تمدید اشتراک‌ها و ارسال اعلان‌ها.

14. \*\*SystemMonitoringService\*\*

- نظارت بر عملکرد سیستم و جمع‌آوری متریک‌ها.

15. \*\*HealthCheckService\*\*

- بررسی سلامت سیستم و منابع (CPU، حافظه، دیسک، شبکه).

16. \*\*Tests\*\*

- شامل تست‌های واحد (Unit Tests) و ادغام (Integration Tests) برای تمام ماژول‌ها.

---

این لیست شامل تمام ماژول‌های پروژه است که به‌صورت کامل و مستقل طراحی شده‌اند و می‌توانند به‌عنوان بخشی از یک سیستم ردیابی GPS یا هر سیستم دیگری استفاده شوند.

بله، قصد دارم تمام جزئیات \*\*SharedKernel\*\* را به‌صورت کاملاً واضح و ساختارمند توضیح دهم. این ماژول یکی از مهم‌ترین بخش‌های پروژه است و شامل ابزارها، قراردادها (Interfaces)، استثناء‌ها (Exceptions)، مدل‌ها، و سایر عناصر عمومی است که در سراسر پروژه استفاده می‌شوند. این بخش به‌گونه‌ای طراحی شده است که مستقل از فریم‌ورک‌ها باشد و قابلیت استفاده مجدد بالایی داشته باشد.

---

## \*\*1. هدف SharedKernel\*\*

\*\*SharedKernel\*\* یک ماژول مشترک است که:

- \*\*ابزارهای عمومی:\*\* ابزارهایی که در سراسر پروژه استفاده می‌شوند را فراهم می‌کند.

- \*\*قراردادها (Interfaces):\*\* Interface‌های مشترک برای تعامل بین ماژول‌ها را تعریف می‌کند.

- \*\*مدل‌های پایه:\*\* کلاس‌های پایه مانند `BaseEntity`، `ValueObject`، و `Result<T>` را ارائه می‌دهد.

- \*\*استقلال از فریم‌ورک‌ها:\*\* طراحی شده است که مستقل از فریم‌ورک‌های خاص (مانند ASP.NET Core یا Entity Framework) باشد.

- \*\*مقیاس‌پذیری:\*\* قابلیت گسترش و مقیاس‌پذیری دارد.

---

## \*\*2. ساختار پوشه‌بندی SharedKernel\*\*

```plaintext

/SharedKernel/

├── Abstractions/

│ ├── IRepository.cs

│ ├── IAggregateRoot.cs

│ └── IEvent.cs

├── Audit/

│ └── AuditLogService.cs

├── Behaviors/

│ └── LoggingBehavior.cs

├── Common/

│ ├── BaseEntity.cs

│ ├── ValueObject.cs

│ ├── Result.cs

│ └── PaginatedResult.cs

├── Config/

│ └── AppSettings.cs

├── Constants/

│ ├── AppConstants.cs

│ ├── Roles.cs

│ └── ClaimTypes.cs

├── Enums/

│ ├── NotificationType.cs

│ └── CommandType.cs

├── Exceptions/

│ ├── NotFoundException.cs

│ ├── ValidationException.cs

│ └── UnauthorizedException.cs

├── Extensions/

│ ├── ServiceCollectionExtensions.cs

│ └── StringExtensions.cs

├── Interfaces/

│ ├── IService.cs

│ └── IPipelineBehavior.cs

├── Middleware/

│ └── ExceptionHandlingMiddleware.cs

├── Notifications/

│ └── Notification.cs

├── Results/

│ └── OperationResult.cs

├── Specifications/

│ ├── Specification.cs

│ └── PagingSpecification.cs

├── Utilities/

│ └── DateTimeUtility.cs

├── Validation/

│ └── FluentValidationExtensions.cs

├── Resources/

│ └── LocalizationResources.cs

└── SharedKernel.csproj

\*\*3. توضیحات هر بخش\*\*

\*\*3.1. Abstractions (قراردادها)\*\*

\*\*هدف:\*\*

این بخش شامل Interface‌های عمومی است که در سراسر پروژه استفاده می‌شوند. این Interface‌ها به عنوان قراردادهایی عمل می‌کنند که تعامل بین لایه‌ها را تسهیل می‌کنند.

\*\*کدهای نمونه:\*\*

\*\*IRepository.cs\*\*

این Interface برای دسترسی به داده‌ها و مدیریت موجودیت‌ها استفاده می‌شود.

```csharp

public interface IRepository<T> where T : class

{

Task<T?> GetByIdAsync(Guid id);

Task<IEnumerable<T>> GetAllAsync();

Task AddAsync(T entity);

void Update(T entity);

void Delete(T entity);

}

\*\*IAggregateRoot.cs\*\*

این Interface برای مشخص کردن Aggregate Rootها است.

```csharp

public interface IAggregateRoot { }

\*\*IEvent.cs\*\*

این Interface برای رویدادها (Events) استفاده می‌شود.

```csharp

public interface IEvent

{

DateTime OccurredOn { get; }

}

\*\*3.2. Audit (لاگ‌زنی)\*\*

\*\*هدف:\*\*

این بخش برای ثبت لاگ‌های تغییرات (Audit Logs) استفاده می‌شود. این لاگ‌ها به شما کمک می‌کنند تا تمام تغییرات مهم در سیستم را ردیابی کنید.

\*\*کدهای نمونه:\*\*

\*\*AuditLogService.cs\*\*

این کلاس برای ثبت لاگ‌های تغییرات (Audit Logs) استفاده می‌شود.

```csharp

public class AuditLogService

{

public void Log(string action, string entityName, Guid entityId)

{

Console.WriteLine($"Action: {action}, Entity: {entityName}, ID: {entityId}");

}

}

\*\*3.3. Behaviors (رفتارها)\*\*

\*\*هدف:\*\*

این بخش شامل Pipeline Behaviors است که برای مدیریت رفتارهای خاص در زمان اجرای درخواست‌ها استفاده می‌شوند. این رفتارها معمولاً برای لاگ‌زنی، اعتبارسنجی، و مدیریت خطاهای سراسری استفاده می‌شوند.

\*\*کدهای نمونه:\*\*

\*\*LoggingBehavior.cs\*\*

این کلاس برای مدیریت Pipeline Behaviors در MediatR استفاده می‌شود.

```csharp

public class LoggingBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>

{

private readonly ILogger<LoggingBehavior<TRequest, TResponse>> \_logger;

public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger)

{

\_logger = logger;

}

public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)

{

\_logger.LogInformation($"Handling {typeof(TRequest).Name}");

var response = await next();

\_logger.LogInformation($"Handled {typeof(TResponse).Name}");

return response;

}

}

\*\*3.4. Common (کلاس‌های پایه)\*\*

\*\*هدف:\*\*

این بخش شامل کلاس‌های پایه است که در سراسر پروژه استفاده می‌شوند، مانند `BaseEntity`، `ValueObject`، و `Result<T>`.

\*\*کدهای نمونه:\*\*

\*\*BaseEntity.cs\*\*

کلاس پایه برای تمامی موجودیت‌ها.

```csharp

public abstract class BaseEntity

{

public Guid Id { get; protected set; } = Guid.NewGuid();

public DateTime CreatedAt { get; protected set; } = DateTime.UtcNow;

public DateTime? UpdatedAt { get; set; }

}

\*\*ValueObject.cs\*\*

کلاس پایه برای Value Objects.

```csharp

public abstract class ValueObject

{

protected abstract IEnumerable<object?> GetEqualityComponents();

public override bool Equals(object? obj)

{

if (obj == null || obj.GetType() != GetType()) return false;

return GetEqualityComponents().SequenceEqual(((ValueObject)obj).GetEqualityComponents());

}

public override int GetHashCode()

{

return GetEqualityComponents()

.Aggregate(1, (current, obj) => current \* 23 + (obj?.GetHashCode() ?? 0));

}

}

\*\*Result.cs\*\*

کلاس برای مدیریت نتایج عملیات.

```csharp

public class Result<T>

{

public bool IsSuccess { get; set; }

public string? Error { get; set; }

public T? Data { get; set; }

public static Result<T> Success(T data) => new() { IsSuccess = true, Data = data };

public static Result<T> Failure(string error) => new() { IsSuccess = false, Error = error };

}

\*\*PaginatedResult.cs\*\*

کلاس برای مدیریت نتایج صفحه‌بندی.

```csharp

public record PaginatedResult<T>(IEnumerable<T> Items, int TotalCount, int PageNumber, int PageSize)

{

public int TotalPages => (int)Math.Ceiling((double)TotalCount / PageSize);

}

\*\*3.5. Config (تنظیمات پیکربندی)\*\*

\*\*هدف:\*\*

این بخش شامل تنظیمات اختصاصی برنامه است که معمولاً از فایل‌هایی مانند `appsettings.json` خوانده می‌شوند.

کدهای نمونه:\*\*

AppSettings.cs\*\*

این کلاس برای مدیریت تنظیمات برنامه استفاده می‌شود.

```csharp

public class AppSettings

{

public string ConnectionString { get; set; }

public JwtSettings JwtSettings { get; set; }

}

public class JwtSettings

{

public string Key { get; set; }

public string Issuer { get; set; }

public string Audience { get; set; }

}

3.6. Constants (ثوابت)\*\*

هدف:\*\*

این بخش شامل مقادیر ثابت است که در سراسر پروژه استفاده می‌شوند.

\*\*کدهای نمونه:\*\*

Roles.cs\*\*

مقادیر ثابت برای نقش‌ها.

```csharp

public static class Roles

{

public const string Admin = "Admin";

public const string User = "User";

}

ClaimTypes.cs\*\*

مقادیر ثابت برای Claims.

```csharp

public static class CustomClaimTypes

{

public const string UserId = "uid";

public const string DeviceId = "did";

}

3.7. Exceptions (استثناء‌ها)\*\*

هدف:\*\*

این بخش شامل استثناء‌های سفارشی است که برای مدیریت خطاهای مختلف استفاده می‌شوند.

کدهای نمونه:\*\*

NotFoundException.cs\*\*

استثناء برای زمانی که آیتمی پیدا نشود.

```csharp

public class NotFoundException : Exception

{

public NotFoundException(string message) : base(message) { }

}

ValidationException.cs\*\*

استثناء برای خطاهای اعتبارسنجی.

```csharp

public class ValidationException : Exception

{

public ValidationException(string message) : base(message) { }

}

UnauthorizedException.cs\*\*

استثناء برای دسترسی غیرمجاز.

```csharp

public class UnauthorizedException : Exception

{

public UnauthorizedException(string message) : base(message) { }

}

3.8. Extensions (افزونه‌ها)\*\*

هدف:\*\*

این بخش شامل افزونه‌هایی است که به کلاس‌ها یا Interface‌ها قابلیت‌های جدیدی اضافه می‌کنند.

کدهای نمونه:\*\*

ServiceCollectionExtensions.cs\*\*

تنظیمات Dependency Injection.

```csharp

public static class ServiceCollectionExtensions

{

public static IServiceCollection AddSharedKernel(this IServiceCollection services)

{

services.AddScoped<ExceptionHandlingMiddleware>();

return services;

}

}

StringExtensions.cs\*\*

افزونه‌هایی برای Stringها.

```csharp

public static class StringExtensions

{

public static bool IsNullOrWhiteSpace(this string str) => string.IsNullOrWhiteSpace(str);

}

3.9. Middleware (میان‌افزارها)\*\*

هدف:\*\*

این بخش شامل میان‌افزارها (Middlewares) است که برای مدیریت استثناء‌ها، لاگ‌زنی، و سایر وظایف سراسری استفاده می‌شوند.

کدهای نمونه:\*\*

ExceptionHandlingMiddleware.cs\*\*

مدیریت استثناء‌ها در سطح سیستم.

```csharp

public class ExceptionHandlingMiddleware

{

private readonly RequestDelegate \_next;

private readonly ILogger<ExceptionHandlingMiddleware> \_logger;

public ExceptionHandlingMiddleware(RequestDelegate next, ILogger<ExceptionHandlingMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

public async Task Invoke(HttpContext context)

{

try

{

await \_next(context);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Unhandled Exception");

context.Response.StatusCode = 500;

await context.Response.WriteAsync("Internal Server Error");

}

}

}

3.10. Notifications\*\*

هدف:\*\*

این بخش برای مدیریت پیام‌های سیستم استفاده می‌شود.

کدهای نمونه:\*\*

Notification.cs\*\*

کلاس برای پیام‌های سیستم.

```csharp

public record Notification(string Message, string Type = "Info");

3.11. Specifications\*\*

هدف:\*\*

این بخش شامل الگوی Specification است که برای طراحی‌های پیچیده‌تر استفاده می‌شود.

کدهای نمونه:\*\*

Specification.cs\*\*

الگوی Specification برای طراحی‌های پیچیده‌تر.

csharp

public abstract class Specification<T>

{

public abstract Expression<Func<T, bool>> ToExpression();

}

PagingSpecification.cs\*\*

Specification برای صفحه‌بندی.

```csharp

public class PagingSpecification<T> : Specification<T>

{

private readonly int \_pageNumber;

private readonly int \_pageSize;

public PagingSpecification(int pageNumber, int pageSize)

{

\_pageNumber = pageNumber;

\_pageSize = pageSize;

}

public override Expression<Func<T, bool>> ToExpression() => x => true;

}

\*\*نتیجه‌گیری\*\*

\*\*SharedKernel\*\* به‌عنوان هسته مشترک پروژه، شامل تمام ابزارها، قراردادها، و مدل‌های پایه است که در سراسر پروژه استفاده می‌شوند. این ماژول به‌گونه‌ای طراحی شده است که:

- \*\*مقیاس‌پذیری:\*\* قابلیت گسترش و مقیاس‌پذیری دارد.

- \*\*استقلال:\*\* مستقل از فریم‌ورک‌ها و زیرساخت‌ها است.

- \*\*تست‌پذیری:\*\* کد قابل تست و حفظ است.

بله، در ادامه کلاس \*\*`IRepository.cs`\*\* را به‌صورت کامل و همراه با کامنت‌های فارسی برای شما می‌نویسم. این کلاس در پوشه \*\*`Abstractions`\*\* قرار دارد و به‌عنوان یک Interface عمومی برای دسترسی به داده‌ها و مدیریت موجودیت‌ها استفاده می‌شود.

فایل: IRepository.cs\*\*

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace SharedKernel.Abstractions

{

/// <summary>

/// این Interface پایه‌ای برای Repositoryها است که عملیات CRUD را تعریف می‌کند.

/// این Interface مستقل از نوع دیتابیس یا ORM است و می‌تواند در هر لایه‌ای از برنامه استفاده شود.

/// </summary>

/// <typeparam name="T">نوع موجودیت (Entity) که باید مدیریت شود.</typeparam>

public interface IRepository<T> where T : class

{

/// <summary>

/// این متد برای دریافت یک موجودیت بر اساس شناسه (ID) استفاده می‌شود.

/// </summary>

/// <param name="id">شناسه (ID) موجودیت مورد نظر.</param>

/// <returns>موجودیت مورد نظر یا null اگر موجودیت پیدا نشود.</returns>

Task<T?> GetByIdAsync(Guid id);

/// <summary>

/// این متد برای دریافت تمام موجودیت‌های موجود استفاده می‌شود.

/// </summary>

/// <returns>لیستی از تمام موجودیت‌ها.</returns>

Task<IEnumerable<T>> GetAllAsync();

/// <summary>

/// این متد برای اضافه کردن یک موجودیت جدید به دیتابیس استفاده می‌شود.

/// </summary>

/// <param name="entity">موجودیتی که باید اضافه شود.</param>

Task AddAsync(T entity);

/// <summary>

/// این متد برای به‌روزرسانی یک موجودیت موجود استفاده می‌شود.

/// </summary>

/// <param name="entity">موجودیتی که باید به‌روزرسانی شود.</param>

void Update(T entity);

/// <summary>

/// این متد برای حذف یک موجودیت از دیتابیس استفاده می‌شود.

/// </summary>

/// <param name="entity">موجودیتی که باید حذف شود.</param>

void Delete(T entity);

// متدهای اضافی برای عملیات پیچیده‌تر

Task<IEnumerable<T>> GetByFilterAsync(Func<T, bool> predicate);

Task<IEnumerable<T>> GetPagedAsync(int pageNumber, int pageSize);

Task ExecuteTransactionAsync(Func<Task> operations);

// متد برای پشتیبانی از عملیات سفارشی‌تر

IQueryable<T> Query();

}

}

توضیحات کلی\*\*

1. \*\*هدف کلاس:\*\*

- این Interface به‌عنوان یک قرارداد (Contract) برای Repositoryها عمل می‌کند.

- شامل عملیات‌های پایه‌ای مانند خواندن (`GetByIdAsync`, `GetAllAsync`)، اضافه کردن (`AddAsync`)، به‌روزرسانی (`Update`) و حذف (`Delete`) است.

2. \*\*Generic Type (`T`):\*\*

- از جنریک (`T`) برای این استفاده شده است که این Interface برای انواع مختلف موجودیت‌ها قابل استفاده باشد.

- شرط `where T : class` تضمین می‌کند که فقط کلاس‌ها می‌توانند به‌عنوان نوع `T` استفاده شوند.

3. \*\*Asynchronous Methods:\*\*

- تمام متدهایی که ممکن است زمان‌بر باشند (مثل خواندن از دیتابیس) به‌صورت Asynchronous طراحی شده‌اند تا عملکرد بهتری در سیستم‌های مقیاس‌پذیر داشته باشند.

4. \*\*Comment‌های فارسی:\*\*

- تمام متدها و پارامترها با توضیحات فارسی کامنت‌گذاری شده‌اند تا قابل فهم برای توسعه‌دهندگان فارسی‌زبان باشند.

نحوه استفاده در پروژه\*\*

این Interface معمولاً در لایه \*\*Infrastructure\*\* پیاده‌سازی می‌شود. برای مثال:

```csharp

public class UserRepository : IRepository<User>

{

private readonly DbContext \_context;

public UserRepository(DbContext context)

{

\_context = context;

}

public async Task<User?> GetByIdAsync(Guid id)

{

return await \_context.Users.FindAsync(id);

}

public async Task<IEnumerable<User>> GetAllAsync()

{

return await \_context.Users.ToListAsync();

}

public async Task AddAsync(User entity)

{

await \_context.Users.AddAsync(entity);

await \_context.SaveChangesAsync();

}

public void Update(User entity)

{

\_context.Users.Update(entity);

\_context.SaveChanges();

}

public void Delete(User entity)

{

\_context.Users.Remove(entity);

\_context.SaveChanges();

}

}

مزایای استفاده از این Interface\*\*

1. \*\*جدا کردن منطق دسترسی به داده‌ها:\*\*

- این Interface به شما کمک می‌کند تا منطق دسترسی به داده‌ها را از بقیه لایه‌ها جدا کنید.

2. \*\*آزمایش‌پذیری (Testability):\*\*

- با استفاده از این Interface می‌توانید تست‌های واحد (Unit Tests) بنویسید و از Mock Objects استفاده کنید.

3. \*\*مقیاس‌پذیری:\*\*

- این Interface به‌راحتی می‌تواند برای انواع مختلف دیتابیس‌ها یا ORMها پیاده‌سازی شود.

4. \*\*استقلال از فریم‌ورک:\*\*

- این Interface مستقل از هر فریم‌ورک خاصی (مثل Entity Framework یا Dapper) است و می‌تواند در هر محیطی استفاده شود.

بله، در ادامه کلاس \*\*`IAggregateRoot.cs`\*\* را به‌صورت کامل و همراه با کامنت‌های فارسی برای شما می‌نویسم. این Interface یکی از قراردادهای مهم در طراحی DDD (Domain-Driven Design) است و به‌عنوان پایه‌ای برای Aggregate Roots (ریشه‌های تجمیع) در دامنه (Domain) استفاده می‌شود.

فایل: IAggregateRoot.cs\*\*

```csharp

using System;

namespace SharedKernel.Abstractions

{

/// <summary>

/// این Interface به‌عنوان قرارداد برای Aggregate Roots استفاده می‌شود.

/// Aggregate Rootها موجودیت‌هایی هستند که به‌عنوان نقطه ورود برای دسترسی به سایر موجودیت‌ها و Value Objects عمل می‌کنند.

/// </summary>

public interface IAggregateRoot

{

/// <summary>

/// شناسه منحصر به فرد Aggregate Root.

/// این شناسه باید در تمام موجودیت‌ها یکتا باشد.

/// </summary>

Guid Id { get; }

/// <summary>

/// زمان ایجاد Aggregate Root.

/// این فیلد نشان‌دهنده زمان ایجاد اولیه موجودیت است.

/// </summary>

DateTime CreatedAt { get; }

/// <summary>

/// زمان آخرین به‌روزرسانی Aggregate Root.

/// این فیلد نشان‌دهنده زمان آخرین تغییر در موجودیت است.

/// </summary>

DateTime? UpdatedAt { get; }

/// <summary>

/// لیست رویدادهای دامنه (Domain Events) که در اثر تغییرات در Aggregate Root ایجاد شده‌اند.

/// این رویدادها معمولاً برای انتشار به سایر بخش‌های سیستم استفاده می‌شوند.

/// </summary>

IReadOnlyCollection<IDomainEvent> DomainEvents { get; }

/// <summary>

/// اضافه کردن یک رویداد دامنه به لیست رویدادهای Aggregate Root.

/// </summary>

/// <param name="domainEvent">رویداد دامنه که باید اضافه شود.</param>

void AddDomainEvent(IDomainEvent domainEvent);

/// <summary>

/// حذف یک رویداد دامنه از لیست رویدادهای Aggregate Root.

/// </summary>

/// <param name="domainEvent">رویداد دامنه که باید حذف شود.</param>

void RemoveDomainEvent(IDomainEvent domainEvent);

/// <summary>

/// پاک کردن تمام رویدادهای دامنه از لیست رویدادهای Aggregate Root.

/// </summary>

void ClearDomainEvents();

}

}

توضیحات کلی\*\*

1. \*\*هدف کلاس:\*\*

- این Interface به‌عنوان قرارداد برای Aggregate Roots استفاده می‌شود.

- Aggregate Roots نقش مهمی در طراحی DDD دارند و به‌عنوان نقطه ورود برای دسترسی به سایر موجودیت‌ها و Value Objects عمل می‌کنند.

2. \*\*ویژگی‌های کلیدی:\*\*

- \*\*`Id`:\*\* شناسه منحصر به فرد Aggregate Root.

- \*\*`CreatedAt`:\*\* زمان ایجاد Aggregate Root.

- \*\*`UpdatedAt`:\*\* زمان آخرین به‌روزرسانی Aggregate Root.

- \*\*`DomainEvents`:\*\* لیست رویدادهای دامنه (Domain Events) که در اثر تغییرات در Aggregate Root ایجاد شده‌اند.

3. \*\*متدهای کلیدی:\*\*

- \*\*`AddDomainEvent`:\*\* اضافه کردن یک رویداد دامنه به لیست رویدادهای Aggregate Root.

- \*\*`RemoveDomainEvent`:\*\* حذف یک رویداد دامنه از لیست رویدادهای Aggregate Root.

- \*\*`ClearDomainEvents`:\*\* پاک کردن تمام رویدادهای دامنه از لیست رویدادهای Aggregate Root.

4. \*\*Comment‌های فارسی:\*\*

- تمام متدها و ویژگی‌ها با توضیحات فارسی کامنت‌گذاری شده‌اند تا قابل فهم برای توسعه‌دهندگان فارسی‌زبان باشند.

نحوه استفاده در پروژه\*\*

این Interface معمولاً توسط موجودیت‌هایی که نقش Aggregate Root را ایفا می‌کنند پیاده‌سازی می‌شود. برای مثال:

```csharp

public class OrderEntity : BaseEntity, IAggregateRoot

{

private readonly List<IDomainEvent> \_domainEvents = new();

public IReadOnlyCollection<IDomainEvent> DomainEvents => \_domainEvents.AsReadOnly();

public void AddDomainEvent(IDomainEvent domainEvent)

{

\_domainEvents.Add(domainEvent);

}

public void RemoveDomainEvent(IDomainEvent domainEvent)

{

\_domainEvents.Remove(domainEvent);

}

public void ClearDomainEvents()

{

\_domainEvents.Clear();

}

public void PlaceOrder()

{

// منطق ایجاد سفارش

AddDomainEvent(new OrderPlacedEvent(this.Id));

}

}

مزایای استفاده از این Interface\*\*

1. \*\*جدا کردن منطق Aggregate Roots:\*\*

- این Interface به شما کمک می‌کند تا منطق Aggregate Roots را از بقیه لایه‌ها جدا کنید.

2. \*\*مدیریت رویدادهای دامنه:\*\*

- این Interface شامل متدهایی برای مدیریت رویدادهای دامنه است که در اثر تغییرات در Aggregate Root ایجاد می‌شوند.

3. \*\*آزمایش‌پذیری (Testability):\*\*

- با استفاده از این Interface می‌توانید تست‌های واحد (Unit Tests) بنویسید و از Mock Objects استفاده کنید.

4. \*\*مقیاس‌پذیری:\*\*

- این Interface به‌راحتی می‌تواند برای انواع مختلف Aggregate Roots پیاده‌سازی شود.

5. \*\*استقلال از فریم‌ورک:\*\*

- این Interface مستقل از هر فریم‌ورک خاصی (مثل Entity Framework یا Dapper) است و می‌تواند در هر محیطی استفاده شود.

در ادامه، کد کامل و بهینه‌شده برای کلاس `AuditLogService` با تمامی جزئیات و کامنت‌های فارسی ارائه می‌شود. این کلاس شامل قابلیت ثبت تغییرات در دیتابیس با استفاده از یک Repository و Helper است.

AuditLogService.cs\*\*

```csharp

using System.Collections.Generic;

using System.Threading.Tasks;

namespace SharedKernel.Audit

{

/// <summary>

/// سرویس لاگ‌گیری تغییرات.

/// </summary>

public class AuditLogService

{

private readonly Repositories.IAuditRepository \_auditRepository;

/// <summary>

/// سازنده سرویس لاگ‌گیری.

/// </summary>

/// <param name="auditRepository">Repository برای ذخیره لاگ‌ها.</param>

public AuditLogService(Repositories.IAuditRepository auditRepository)

{

// تزریق وابستگی IAuditRepository برای ذخیره لاگ‌ها در دیتابیس

\_auditRepository = auditRepository;

}

/// <summary>

/// ثبت تغییرات در دیتابیس.

/// </summary>

/// <typeparam name="T">نوع موجودیت (Entity).</typeparam>

/// <param name="oldEntity">نسخه قدیمی موجودیت (برای عملیات Update و Delete).</param>

/// <param name="newEntity">نسخه جدید موجودیت (برای عملیات Create و Update).</param>

/// <param name="changedBy">شناسه کاربر عامل.</param>

/// <param name="entityId">شناسه موجودیت.</param>

/// <param name="operationType">نوع عملیات (Create, Update, Delete).</param>

/// <param name="ipAddress">آدرس IP کاربر (اختیاری).</param>

/// <param name="userAgent">User-Agent کاربر (اختیاری).</param>

/// <param name="reason">دلیل تغییر (اختیاری).</param>

public async Task LogChangesAsync<T>(

T oldEntity,

T newEntity,

string changedBy,

string entityId,

string operationType,

string ipAddress = null,

string userAgent = null,

string reason = null)

{

// استفاده از AuditTrailHelper برای استخراج تغییرات بین نسخه قدیمی و جدید

var changes = Helpers.AuditTrailHelper.GetAuditChanges(

oldEntity,

newEntity,

changedBy,

entityId,

operationType,

ipAddress,

userAgent,

reason);

// ذخیره تغییرات در دیتابیس

await \_auditRepository.SaveAsync(changes);

}

}

}

توضیحات کد\*\*

1. \*\*Dependency Injection:\*\*

- کلاس `AuditLogService` از طریق Constructor Injection به `IAuditRepository` وابسته است. این رویکرد باعث می‌شود که کلاس قابل تست و انعطاف‌پذیر باشد.

2. \*\*Generic Type (`<T>`):\*\*

- این کلاس از Generic Type استفاده می‌کند تا بتواند برای هر نوع موجودیتی (Entity) استفاده شود. این ویژگی باعث می‌شود که کد قابل استفاده در سناریوهای مختلف باشد.

3. \*\*استفاده از `AuditTrailHelper`:\*\*

- کلاس `AuditTrailHelper` برای مقایسه نسخه قدیمی و جدید موجودیت‌ها استفاده می‌شود. این کلاس تغییرات را به صورت لیستی از `AuditChange` برمی‌گرداند.

4. \*\*پارامترهای اختیاری:\*\*

- پارامترهای `ipAddress`, `userAgent`, و `reason` به صورت اختیاری هستند و می‌توانند برای نیازهای خاص مانند ردیابی دقیق‌تر کاربر یا ذخیره دلیل تغییر استفاده شوند.

5. \*\*ذخیره در دیتابیس:\*\*

- تغییرات استخراج‌شده توسط `AuditTrailHelper` به صورت آسنکرون در دیتابیس ذخیره می‌شوند. این کار توسط متد `SaveAsync` از `IAuditRepository` انجام می‌شود.

\*\*مثال استفاده\*\*

فرض کنید یک موجودیت `User` دارید که می‌خواهید تغییرات آن را لاگ کنید:

```csharp

public class UserService

{

private readonly AuditLogService \_auditLogService;

public UserService(AuditLogService auditLogService)

{

\_auditLogService = auditLogService;

}

public async Task CreateUserAsync(User newUser, string userId, string ipAddress, string userAgent)

{

// ثبت لاگ برای عملیات Create

await \_auditLogService.LogChangesAsync(

null,

newUser,

"Admin",

userId.ToString(),

"Create",

ipAddress,

userAgent,

"ایجاد کاربر جدید");

}

public async Task UpdateUserAsync(User oldUser, User updatedUser, string userId, string ipAddress, string userAgent)

{

// ثبت لاگ برای عملیات Update

await \_auditLogService.LogChangesAsync(

oldUser,

updatedUser,

"Admin",

userId.ToString(),

"Update",

ipAddress,

userAgent,

"به‌روزرسانی اطلاعات کاربر");

}

}

public class User

{

public Guid Id { get; set; }

public string Name { get; set; }

public string Email { get; set; }

}

\*\*مزایای این طراحی\*\*

1. \*\*ماژولار بودن:\*\*

- کلاس `AuditLogService` به طور کامل از `IAuditRepository` و `AuditTrailHelper` وابسته است و هیچ منطق خاصی در خود ندارد. این باعث می‌شود که قابلیت تست و توسعه آن بالا باشد.

2. \*\*قابلیت استفاده مجدد:\*\*

- به دلیل استفاده از Generic Type و Dependency Injection، این کلاس می‌تواند برای هر موجودیتی استفاده شود.

3. \*\*لاگ‌گیری دقیق:\*\*

- تمامی اطلاعات مرتبط با تغییرات (مانند IP Address، User-Agent، و دلیل تغییر) ثبت می‌شوند که برای مانیتورینگ و تحلیل بعدی مفید است.

4. \*\*پشتیبانی از Async:\*\*

- تمامی متدها به صورت آسنکرون هستند که باعث بهبود عملکرد در محیط‌های با حجم بالا می‌شود.

در ادامه، کدهای کامل و بهینه‌شده برای دو کلاس `AuditChange` و `AuditTrailHelper` با کامنت‌های فارسی نوشته شده‌اند. این کدها شامل توضیحات دقیق و کامل برای هر بخش هستند.

---

### \*\*AuditChange.cs\*\*

```csharp

using System;

using System.Text.Json;

namespace SharedKernel.Audit.Models

{

/// <summary>

/// مدل داده‌ای برای ثبت تغییرات در دیتابیس.

/// </summary>

public class AuditChange

{

/// <summary>

/// شناسه یکتا لاگ تغییرات.

/// </summary>

public Guid Id { get; set; }

/// <summary>

/// نوع عملیات (Create, Update, Delete).

/// </summary>

public AuditOperationType OperationType { get; set; }

/// <summary>

/// نام موجودیت (Entity) که تغییر در آن انجام شده است.

/// </summary>

public string EntityName { get; set; }

/// <summary>

/// شناسه موجودیت (Entity ID) که تغییر در آن انجام شده است.

/// </summary>

public string EntityId { get; set; }

/// <summary>

/// نام فیلد (با dot notation برای Nested Objects).

/// </summary>

public string PropertyName { get; set; }

/// <summary>

/// مقدار قبلی فیلد (به صورت JSON ذخیره می‌شود).

/// </summary>

public JsonElement OldValue { get; set; }

/// <summary>

/// مقدار جدید فیلد (به صورت JSON ذخیره می‌شود).

/// </summary>

public JsonElement NewValue { get; set; }

/// <summary>

/// شناسه کاربر عامل.

/// </summary>

public string ChangedBy { get; set; }

/// <summary>

/// IP Address کاربر.

/// </summary>

public string IpAddress { get; set; }

/// <summary>

/// User-Agent کاربر.

/// </summary>

public string UserAgent { get; set; }

/// <summary>

/// تاریخ و زمان تغییر.

/// </summary>

public DateTime ChangedAt { get; set; }

/// <summary>

/// دلیل تغییر (اختیاری).

/// </summary>

public string ChangeReason { get; set; }

}

/// <summary>

/// انواع عملیات لاگ‌گیری.

/// </summary>

public enum AuditOperationType

{

/// <summary>

/// ایجاد رکورد جدید.

/// </summary>

Create,

/// <summary>

/// به‌روزرسانی رکورد موجود.

/// </summary>

Update,

/// <summary>

/// حذف رکورد.

/// </summary>

Delete

}

}

```

---

### \*\*AuditTrailHelper.cs\*\*

```csharp

using SharedKernel.Audit.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Text.Json;

namespace SharedKernel.Audit.Helpers

{

/// <summary>

/// کلاس کمکی برای تشخیص تغییرات بین نسخه قدیمی و جدید موجودیت‌ها.

/// </summary>

public static class AuditTrailHelper

{

/// <summary>

/// مقایسه دو موجودیت و استخراج تغییرات.

/// </summary>

/// <typeparam name="T">نوع موجودیت (Entity).</typeparam>

/// <param name="oldEntity">نسخه قدیمی موجودیت (برای عملیات Update و Delete).</param>

/// <param name="newEntity">نسخه جدید موجودیت (برای عملیات Create و Update).</param>

/// <param name="changedBy">شناسه کاربر عامل.</param>

/// <param name="entityId">شناسه موجودیت.</param>

/// <param name="operationType">نوع عملیات (Create, Update, Delete).</param>

/// <param name="ipAddress">آدرس IP کاربر (اختیاری).</param>

/// <param name="userAgent">User-Agent کاربر (اختیاری).</param>

/// <param name="reason">دلیل تغییر (اختیاری).</param>

/// <returns>لیستی از تغییرات استخراج‌شده.</returns>

public static List<AuditChange> GetAuditChanges<T>(

T oldEntity,

T newEntity,

string changedBy,

string entityId,

AuditOperationType operationType,

string ipAddress = null,

string userAgent = null,

string reason = null)

{

var changes = new List<AuditChange>();

// ذخیره نام موجودیت برای استفاده در لاگ‌ها

string entityName = typeof(T).Name;

// لیست سیاه برای نادیده گرفتن پراپرتی‌های معروف

var excludedProps = new[] { "UpdatedAt", "LastModified", "Timestamp" };

// دریافت تمامی پراپرتی‌های موجودیت

var properties = typeof(T).GetProperties(BindingFlags.Public | BindingFlags.Instance);

foreach (var prop in properties)

{

// اگر پراپرتی باید از لاگ‌گیری حذف شود، رد می‌شود

if (!prop.CanRead ||

prop.GetCustomAttributes(typeof(IgnoreAuditAttribute), false).Any() ||

excludedProps.Contains(prop.Name))

continue;

// دریافت مقادیر قدیمی و جدید پراپرتی

var oldValue = SafeSerialize(oldEntity != null ? prop.GetValue(oldEntity) : null);

var newValue = SafeSerialize(newEntity != null ? prop.GetValue(newEntity) : null);

// بررسی نوع عملیات و ثبت تغییرات متناظر

if (operationType == AuditOperationType.Create && !string.IsNullOrEmpty(newValue))

{

changes.Add(CreateAuditRecord(prop.Name, oldValue, newValue, changedBy, entityId, operationType, ipAddress, userAgent, reason, entityName));

}

else if (operationType == AuditOperationType.Delete && !string.IsNullOrEmpty(oldValue))

{

changes.Add(CreateAuditRecord(prop.Name, oldValue, newValue, changedBy, entityId, operationType, ipAddress, userAgent, reason, entityName));

}

else if (oldValue != newValue)

{

changes.Add(CreateAuditRecord(prop.Name, oldValue, newValue, changedBy, entityId, operationType, ipAddress, userAgent, reason, entityName));

}

}

return changes;

}

/// <summary>

/// ایجاد یک رکورد لاگ تغییرات.

/// </summary>

private static AuditChange CreateAuditRecord(

string propertyName,

string oldValue,

string newValue,

string changedBy,

string entityId,

AuditOperationType operationType,

string ipAddress,

string userAgent,

string reason,

string entityName)

{

return new AuditChange

{

Id = Guid.NewGuid(), // شناسه یکتا لاگ تغییرات

OperationType = operationType, // نوع عملیات

EntityName = entityName, // نام موجودیت

EntityId = entityId, // شناسه موجودیت

PropertyName = propertyName, // نام پراپرتی

OldValue = ParseToJsonElement(oldValue), // مقدار قبلی (JSON)

NewValue = ParseToJsonElement(newValue), // مقدار جدید (JSON)

ChangedBy = changedBy, // شناسه کاربر عامل

IpAddress = ipAddress, // آدرس IP کاربر

UserAgent = userAgent, // User-Agent کاربر

ChangedAt = DateTime.UtcNow, // تاریخ و زمان تغییر

ChangeReason = reason // دلیل تغییر

};

}

/// <summary>

/// سریالایز امن برای مقادیر پراپرتی‌ها.

/// </summary>

private static string SafeSerialize(object value)

{

try

{

// اگر مقدار Null باشد، به صورت خالی ذخیره شود

if (value == null) return "null";

// اگر پراپرتی دارای Attribute [MaskAudit] باشد، مقدار ماسک شود

var type = value.GetType();

if (type.GetCustomAttributes(typeof(MaskAuditAttribute), false).Any())

return "\"\*\*\*\*\*\"";

// سریالایز مقادیر به JSON

return JsonSerializer.Serialize(value);

}

catch

{

// در صورت بروز خطای سریالایز، مقدار "[Unserializable]" ذخیره شود

return "\"[Unserializable]\"";

}

}

/// <summary>

/// تبدیل رشته JSON به JsonElement.

/// </summary>

private static JsonElement ParseToJsonElement(string value)

{

if (string.IsNullOrEmpty(value))

return default;

using var document = JsonDocument.Parse(value);

return document.RootElement.Clone();

}

}

/// <summary>

/// Attribute برای ماسک کردن مقادیر حساس.

/// </summary>

[AttributeUsage(AttributeTargets.Property)]

public class MaskAuditAttribute : Attribute

{

}

/// <summary>

/// Attribute برای نادیده گرفتن پراپرتی‌های خاص.

/// </summary>

[AttributeUsage(AttributeTargets.Property)]

public class IgnoreAuditAttribute : Attribute

{

}

}

\*\*توضیحات کلی\*\*

1. \*\*مدل `AuditChange`:\*\*

- این مدل برای ذخیره تغییرات در دیتابیس طراحی شده است.

- فیلدهای `OldValue` و `NewValue` به صورت `JsonElement` ذخیره می‌شوند تا بتوان مقادیر پیچیده مثل آبجکت‌ها و لیست‌ها را نیز ذخیره کرد.

2. \*\*کلاس `AuditTrailHelper`:\*\*

- این کلاس برای مقایسه نسخه قدیمی و جدید موجودیت‌ها و استخراج تغییرات استفاده می‌شود.

- از Reflection برای دریافت پراپرتی‌های موجودیت‌ها استفاده می‌کند.

- از `SafeSerialize` برای سریالایز امن مقادیر استفاده می‌کند.

- از `ParseToJsonElement` برای تبدیل رشته‌های JSON به `JsonElement` استفاده می‌کند.

3. \*\*Attributes:\*\*

- `[MaskAudit]` برای ماسک کردن مقادیر حساس استفاده می‌شود.

- `[IgnoreAudit]` برای نادیده گرفتن پراپرتی‌های خاص استفاده می‌شود.

4. \*\*Enum `AuditOperationType`:\*\*

- این Enum برای تعیین نوع عملیات (`Create`, `Update`, `Delete`)

با توجه به پیشنهادات ارائه‌شده، کد کامل و بهینه‌شده برای کلاس `IgnoreAuditAttribute` و متد Extension مرتبط با آن به شرح زیر خواهد بود. این کد شامل تمامی ویژگی‌ها و بهبودهای پیشنهادی است.

\*\*IgnoreAuditAttribute.cs\*\*

```csharp

using System;

namespace SharedKernel.Audit.Attributes

{

/// <summary>

/// Attribute برای مشخص کردن فیلدهایی که نباید در لاگ‌گیری ثبت شوند.

/// </summary>

/// <remarks>

/// این Attribute روی پراپرتی‌هایی قرار می‌گیرد که نمی‌خواهید در فرآیند لاگ‌گیری تغییرات، مقادیر آن‌ها ذخیره شود.

/// به عنوان مثال، پراپرتی‌های حساس مثل "Password" یا پراپرتی‌های غیرضروری مثل "UpdatedAt" می‌توانند با این Attribute نادیده گرفته شوند.

/// </remarks>

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]

public class IgnoreAuditAttribute : Attribute

{

/// <summary>

/// دلیل نادیده گرفتن پراپرتی (اختیاری).

/// </summary>

public string Reason { get; }

/// <summary>

/// سازنده پیش‌فرض کلاس.

/// </summary>

/// <param name="reason">دلیل نادیده گرفتن پراپرتی (اختیاری).</param>

public IgnoreAuditAttribute(string reason = "")

{

Reason = reason;

}

}

}

\*\*PropertyInfoExtensions.cs\*\*

```csharp

using System;

using System.Reflection;

using SharedKernel.Audit.Attributes;

namespace SharedKernel.Audit.Extensions

{

/// <summary>

/// کلاس کمکی برای افزودن متدهای Extension برای PropertyInfo.

/// </summary>

public static class PropertyInfoExtensions

{

/// <summary>

/// بررسی اینکه آیا یک پراپرتی باید در لاگ‌گیری نادیده گرفته شود یا خیر.

/// </summary>

/// <param name="propertyInfo">اطلاعات پراپرتی.</param>

/// <returns>True اگر پراپرتی باید نادیده گرفته شود، در غیر این صورت False.</returns>

public static bool IsIgnoredForAudit(this PropertyInfo propertyInfo)

{

// بررسی وجود Attribute [IgnoreAudit]

return propertyInfo.GetCustomAttributes(typeof(IgnoreAuditAttribute), false).Any();

}

/// <summary>

/// دریافت دلیل نادیده گرفتن پراپرتی (اگر وجود داشته باشد).

/// </summary>

/// <param name="propertyInfo">اطلاعات پراپرتی.</param>

/// <returns>دلیل نادیده گرفتن پراپرتی، اگر وجود داشته باشد؛ در غیر این صورت Null.</returns>

public static string GetIgnoreReason(this PropertyInfo propertyInfo)

{

// دریافت Attribute [IgnoreAudit]

var attribute = propertyInfo.GetCustomAttributes(typeof(IgnoreAuditAttribute), false)

.OfType<IgnoreAuditAttribute>()

.FirstOrDefault();

// بازگرداندن دلیل (اگر وجود داشته باشد)

return attribute?.Reason;

}

}

}

مثال استفاده\*\*

1. استفاده از `IgnoreAuditAttribute`\*\*

```csharp

public class User

{

public Guid Id { get; set; }

public string Name { get; set; }

[IgnoreAudit("Contains sensitive information")]

public string Password { get; set; }

[IgnoreAudit("Auto-generated field")]

public DateTime UpdatedAt { get; set; }

}

\*\*2. استفاده از متدهای Extension\*\*

در کلاس `AuditTrailHelper`، می‌توانید از متدهای Extension برای بررسی و دریافت دلایل نادیده گرفتن پراپرتی‌ها استفاده کنید:

```csharp

foreach (var prop in properties)

{

// اگر پراپرتی باید از لاگ‌گیری حذف شود، رد می‌شود

if (!prop.CanRead || prop.IsIgnoredForAudit())

{

var reason = prop.GetIgnoreReason(); // دریافت دلیل نادیده گرفتن

\_logger.LogInformation($"Property '{prop.Name}' is ignored for audit. Reason: {reason}");

continue;

}

}

\*\*توضیحات کد\*\*

1. \*\*افزودن دلیل نادیده گرفتن (`Reason`):\*\*

- این ویژگی به شما اجازه می‌دهد تا دلیل نادیده گرفتن یک پراپرتی را مشخص کنید. این دلیل می‌تواند در لاگ‌ها یا Debugging مفید باشد.

2. \*\*استفاده از متد Extension (`IsIgnoredForAudit`):\*\*

- این متد به کد تمیزتر و قابل‌تست‌تری منجر می‌شود. به جای نوشتن منطق بررسی Attribute در هر جایی که نیاز دارید، فقط از این متد استفاده می‌کنید.

3. \*\*دریافت دلیل نادیده گرفتن (`GetIgnoreReason`):\*\*

- این متد به شما اجازه می‌دهد تا دلیل نادیده گرفتن پراپرتی را در صورت وجود، دریافت کنید. این ویژگی می‌تواند در لاگ‌گیری دقیق‌تر مفید باشد.

4. \*\*قابلیت ترکیب با Attribute‌های دیگر:\*\*

- با تنظیم `AllowMultiple = false` در `[AttributeUsage]`، مطمئن می‌شوید که فقط یکبار از این Attribute استفاده شود. اگر نیاز به ترکیب با Attribute‌های دیگر دارید، می‌توانید این مقدار را به `true` تغییر دهید.

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- استفاده از متدهای Extension و دلیل نادیده گرفتن، کد را تمیز و قابل‌فهم می‌کند.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدهای Extension به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*لاگ‌گیری دقیق‌تر:\*\*

- با ذخیره دلیل نادیده گرفتن پراپرتی‌ها، می‌توانید در Debugging و لاگ‌گیری اطلاعات دقیق‌تری داشته باشید.

4. \*\*انعطاف‌پذیری:\*\*

- امکان ترکیب با Attribute‌های دیگر و اضافه کردن ویژگی‌های جدید به راحتی امکان‌پذیر است.

در ادامه، کد کامل و بهینه‌شده برای کلاس `AuditExtensions` با تمامی جزئیات و بهبودهای ممکن ارائه شده است. این کد شامل کامنت‌های فارسی دقیق، مدیریت وابستگی‌ها (Dependency Injection)، و پیشنهادات برای بهبود قابلیت گسترش و خوانایی است.

---

### \*\*AuditExtensions.cs\*\*

```csharp

using Microsoft.Extensions.DependencyInjection;

using SharedKernel.Audit;

using SharedKernel.Audit.Repositories;

namespace SharedKernel.Audit.Extensions

{

/// <summary>

/// Extension Methods برای تنظیمات لاگ‌گیری.

/// </summary>

/// <remarks>

/// این کلاس شامل متدهای Extension برای ثبت سرویس‌های مربوط به لاگ‌گیری در DI Container است.

/// از این کلاس برای جداسازی منطق ثبت وابستگی‌ها از فایل‌های اصلی برنامه (مانند `Program.cs`) استفاده می‌شود.

/// </remarks>

public static class AuditExtensions

{

/// <summary>

/// افزودن سرویس‌های لاگ‌گیری به DI Container.

/// </summary>

/// <param name="services">کانتینر Dependency Injection.</param>

/// <returns>IServiceCollection پس از ثبت سرویس‌ها.</returns>

public static IServiceCollection AddAuditServices(this IServiceCollection services)

{

// ثبت مخازن (Repositories) مرتبط با لاگ‌گیری

services.AddScoped<IAuditRepository, AuditRepository>();

// ثبت سرویس‌های لاگ‌گیری

services.AddScoped<AuditLogService>();

// ثبت سرویس‌های جانبی (اگر نیاز باشد)

// services.AddSingleton<ISomeHelperService, SomeHelperService>();

return services;

}

/// <summary>

/// افزودن تنظیمات اختیاری برای لاگ‌گیری.

/// </summary>

/// <param name="services">کانتینر Dependency Injection.</param>

/// <param name="configureOptions">تابعی برای تنظیم گزینه‌های لاگ‌گیری.</param>

/// <returns>IServiceCollection پس از ثبت سرویس‌ها و تنظیمات.</returns>

public static IServiceCollection AddAuditServices(

this IServiceCollection services,

Action<AuditOptions> configureOptions)

{

// افزودن سرویس‌های پایه لاگ‌گیری

services.AddAuditServices();

// ثبت تنظیمات اختیاری

services.Configure(configureOptions);

return services;

}

}

/// <summary>

/// گزینه‌های تنظیمات لاگ‌گیری.

/// </summary>

public class AuditOptions

{

/// <summary>

/// آیا باید لاگ‌گیری برای مقادیر حساس فعال باشد؟

/// </summary>

public bool EnableSensitiveDataLogging { get; set; } = false;

/// <summary>

/// حداکثر تعداد لاگ‌های ذخیره‌شده در حافظه (اختیاری).

/// </summary>

public int MaxLogEntriesInMemory { get; set; } = 1000;

/// <summary>

/// لیست پراپرتی‌هایی که همیشه باید نادیده گرفته شوند (اختیاری).

/// </summary>

public List<string> IgnoredProperties { get; set; } = new List<string>();

}

}

توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس شامل متدهای Extension برای ثبت سرویس‌های مرتبط با لاگ‌گیری در کانتینر Dependency Injection است.

- از این کلاس برای جداسازی منطق ثبت وابستگی‌ها از فایل‌های اصلی برنامه (مانند `Program.cs`) استفاده می‌شود.

2. \*\*ثبت سرویس‌ها:\*\*

- مخازن (`Repositories`) مرتبط با لاگ‌گیری (مانند `AuditRepository`) به صورت Scoped ثبت شده‌اند.

- سرویس‌های لاگ‌گیری (مانند `AuditLogService`) نیز به صورت Scoped ثبت شده‌اند.

3. \*\*تنظیمات اختیاری:\*\*

- متد `AddAuditServices` دوم، امکان تنظیم گزینه‌های اختیاری را فراهم می‌کند.

- این گزینه‌ها شامل فعال‌سازی لاگ‌گیری مقادیر حساس، محدودیت تعداد لاگ‌ها در حافظه، و لیست پراپرتی‌های نادیده‌گرفته‌شده است.

4. \*\*استفاده از `Action<T>` برای تنظیمات:\*\*

- با استفاده از `Action<AuditOptions>`، می‌توانید تنظیمات اختیاری را به صورت پویا در فایل `Program.cs` اعمال کنید.

5. \*\*مثال استفاده:\*\*

1. استفاده از متد پایه:\*\*

```csharp

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuditServices();

```

2. استفاده از متد با تنظیمات اختیاری:\*\*

```csharp

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuditServices(options =>

{

options.EnableSensitiveDataLogging = true;

options.MaxLogEntriesInMemory = 500;

options.IgnoredProperties.Add("Password");

});```

6. \*\*مدیریت وابستگی‌ها:\*\*

- تمامی سرویس‌ها و مخازن به صورت Scoped ثبت شده‌اند تا در هر Request، یک نمونه جدید ایجاد شود.

- اگر نیاز به سرویس‌های Singleton یا Transient دارید، می‌توانید آن‌ها را نیز به کد اضافه کنید.

7. \*\*قابلیت گسترش:\*\*

- این کلاس طوری طراحی شده است که به راحتی قابل گسترش است. می‌توانید سرویس‌های جدید یا تنظیمات بیشتری را به آن اضافه کنید.

مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدهای Extension به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از تنظیمات اختیاری:\*\*

- امکان تنظیم گزینه‌های اختیاری (مانند لاگ‌گیری مقادیر حساس) به کاربران داده شده است.

4. \*\*جدا کردن منطق:\*\*

- منطق ثبت وابستگی‌ها از فایل‌های اصلی برنامه (مانند `Program.cs`) جدا شده است و اصل SRP (Single Responsibility Principle) رعایت شده است.

5. \*\*قابلیت گسترش:\*\*

- امکان اضافه کردن سرویس‌ها یا تنظیمات جدید به راحتی وجود دارد.

در ادامه، کد کامل و بهینه‌شده برای کلاس `LoggingExtensions` ارائه شده است. این کلاس شامل متدهای Extension برای تسهیل لاگ‌گیری و مدیریت خطاهای سیستمی است. تمامی نکات مهم، بهبودها و قابلیت‌های مورد نیاز در طراحی این کلاس در نظر گرفته شده است.

---

### \*\*LoggingExtensions.cs\*\*

```csharp

using System;

using System.Collections.Generic;

using System.Linq;

using Microsoft.Extensions.Logging;

using SharedKernel.Audit.Models;

using SharedKernel.Audit.Interfaces;

namespace SharedKernel.Audit.Extensions

{

/// <summary>

/// کلاس کمکی برای افزودن قابلیت‌های لاگ‌گیری به برنامه.

/// </summary>

/// <remarks>

/// این کلاس شامل متدهای Extension برای تسهیل لاگ‌گیری و مدیریت خطاهای سیستمی است.

/// از این کلاس می‌توانید برای ثبت لاگ‌ها، مدیریت استثناها و ایجاد لاگ‌های ساختاریافته استفاده کنید.

/// </remarks>

public static class LoggingExtensions

{

/// <summary>

/// ثبت یک لاگ اطلاعاتی (Information) با جزئیات تغییرات.

/// </summary>

/// <param name="logger">Instance از ILogger.</param>

/// <param name="auditChanges">لیست تغییراتی که باید لاگ شوند.</param>

/// <param name="message">پیام اختیاری برای لاگ.</param>

public static void LogAuditChanges(this ILogger logger, List<AuditChange> auditChanges, string message = null)

{

if (auditChanges == null || !auditChanges.Any())

{

logger.LogInformation("No audit changes to log.");

return;

}

foreach (var change in auditChanges)

{

logger.LogInformation(

"Audit Log - Entity: {EntityName}, Property: {PropertyName}, OldValue: {OldValue}, NewValue: {NewValue}, ChangedBy: {ChangedBy}, ChangedAt: {ChangedAt}",

change.EntityName,

change.PropertyName,

change.OldValue.ToString(),

change.NewValue.ToString(),

change.ChangedBy,

change.ChangedAt);

if (!string.IsNullOrEmpty(message))

{

logger.LogInformation("Additional Message: {Message}", message);

}

}

}

/// <summary>

/// ثبت یک لاگ خطا (Error) با جزئیات استثنای رخ‌داده.

/// </summary>

/// <param name="logger">Instance از ILogger.</param>

/// <param name="exception">استثنای رخ‌داده.</param>

/// <param name="message">پیام اختیاری برای لاگ.</param>

public static void LogErrorWithDetails(this ILogger logger, Exception exception, string message = null)

{

if (exception == null)

{

logger.LogWarning("No exception provided for error logging.");

return;

}

logger.LogError(

exception,

"An error occurred: {ErrorMessage}. StackTrace: {StackTrace}",

exception.Message,

exception.StackTrace);

if (!string.IsNullOrEmpty(message))

{

logger.LogError("Additional Message: {Message}", message);

}

}

/// <summary>

/// ثبت یک لاگ هشدار (Warning) با جزئیات.

/// </summary>

/// <param name="logger">Instance از ILogger.</param>

/// <param name="warning">پیام هشدار.</param>

public static void LogWarningWithDetails(this ILogger logger, string warning)

{

if (string.IsNullOrEmpty(warning))

{

logger.LogWarning("No warning message provided for logging.");

return;

}

logger.LogWarning("Warning: {WarningMessage}", warning);

}

/// <summary>

/// ثبت یک لاگ موفقیت‌آمیز (Success) با جزئیات.

/// </summary>

/// <param name="logger">Instance از ILogger.</param>

/// <param name="successMessage">پیام موفقیت.</param>

public static void LogSuccess(this ILogger logger, string successMessage)

{

if (string.IsNullOrEmpty(successMessage))

{

logger.LogInformation("No success message provided for logging.");

return;

}

logger.LogInformation("Success: {SuccessMessage}", successMessage);

}

/// <summary>

/// ثبت یک لاگ سفارشی با سطح دلخواه.

/// </summary>

/// <param name="logger">Instance از ILogger.</param>

/// <param name="logLevel">سطح لاگ (Information, Warning, Error, etc.).</param>

/// <param name="message">پیام لاگ.</param>

/// <param name="args">آرگومان‌های اختیاری برای فرمت‌بندی پیام.</param>

public static void LogCustom(this ILogger logger, LogLevel logLevel, string message, params object[] args)

{

if (string.IsNullOrEmpty(message))

{

logger.LogWarning("No message provided for custom logging.");

return;

}

logger.Log(logLevel, message, args);

}

}

}

\*\*توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس شامل متدهای Extension برای تسهیل لاگ‌گیری است.

- از این متدها می‌توانید برای ثبت لاگ‌های اطلاعاتی، خطاها، هشدارها و موفقیت‌آمیز استفاده کنید.

2. \*\*متدهای موجود:\*\*

- \*\*`LogAuditChanges`:\*\* برای ثبت تغییرات لاگ‌گیری (Audit Changes) استفاده می‌شود.

- \*\*`LogErrorWithDetails`:\*\* برای ثبت خطاهای سیستمی با جزئیات استثنای رخ‌داده.

- \*\*`LogWarningWithDetails`:\*\* برای ثبت هشدارها.

- \*\*`LogSuccess`:\*\* برای ثبت عملیات موفقیت‌آمیز.

- \*\*`LogCustom`:\*\* برای ثبت لاگ‌های سفارشی با سطح دلخواه.

3. \*\*بهبودها:\*\*

- \*\*پشتیبانی از پیام‌های اختیاری:\*\* هر متد می‌تواند یک پیام اختیاری دریافت کند که به لاگ اضافه می‌شود.

- \*\*مدیریت استثناها:\*\* در صورت عدم وجود داده‌های لازم، پیام‌های مناسبی به عنوان هشدار ثبت می‌شوند.

- \*\*پشتیبانی از فرمت‌بندی پیام‌ها:\*\* از آرگومان‌های اختیاری برای فرمت‌بندی پیام‌ها استفاده شده است.

4. \*\*مثال استفاده:\*\*

\*\*1. ثبت لاگ تغییرات (Audit Changes):\*\*

```csharp

var auditChanges = new List<AuditChange>

{

new AuditChange

{

EntityName = "User",

PropertyName = "Email",

OldValue = "old@example.com",

NewValue = "new@example.com",

ChangedBy = "Admin",

ChangedAt = DateTime.UtcNow

}

};

\_logger.LogAuditChanges(auditChanges, "User email updated successfully.");

\*\*2. ثبت خطای سیستمی:\*\*

try

{

// کدی که ممکن است خطا داشته باشد

}

catch (Exception ex)

{

\_logger.LogErrorWithDetails(ex, "An unexpected error occurred while processing the request.");

}

3. ثبت هشدار:\*\*

```csharp

\_logger.LogWarningWithDetails("The system is running low on memory.");

\*\*4. ثبت موفقیت‌آمیز:\*\*

\_logger.LogSuccess("The operation was completed successfully.");

\*\*5. ثبت لاگ سفارشی:\*\*

\_logger.LogCustom(LogLevel.Debug, "Custom debug message with parameter: {Parameter}", "Value");

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدهای Extension به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از انعطاف‌پذیری:\*\*

- امکان اضافه کردن لاگ‌های سفارشی با سطح دلخواه وجود دارد.

4. \*\*مدیریت استثناها:\*\*

- در صورت عدم وجود داده‌های لازم، پیام‌های مناسبی به عنوان هشدار ثبت می‌شوند.

5. \*\*قابلیت گسترش:\*\*

- امکان اضافه کردن متدهای جدید به راحتی وجود دارد.

در ادامه، کد کامل و بهینه‌شده برای کلاس `AuditEntry` ارائه شده است. این کلاس شامل تمامی نیازمندی‌ها و بهبودهای ممکن است و طراحی آن به گونه‌ای است که قابلیت توسعه، خوانایی و استفاده در پروژه‌های بزرگ را داشته باشد. تمامی جزئیات و نکات مهم در نظر گرفته شده‌اند.

---

### \*\*AuditEntry.cs\*\*

```csharp

using System;

using System.Text.Json;

namespace SharedKernel.Audit.Models

{

/// <summary>

/// مدل داده‌ای برای ثبت تغییرات سیستمی (Audit Entry).

/// </summary>

public class AuditEntry

{

/// <summary>

/// شناسه یکتا لاگ تغییرات.

/// </summary>

public Guid Id { get; set; } = Guid.NewGuid();

/// <summary>

/// نوع عملیات (Create, Update, Delete).

/// </summary>

public AuditOperationType OperationType { get; set; }

/// <summary>

/// نام موجودیت (Entity) که تغییر در آن انجام شده است.

/// </summary>

public string EntityName { get; set; }

/// <summary>

/// شناسه موجودیت (Entity ID) که تغییر در آن انجام شده است.

/// </summary>

public string EntityId { get; set; }

/// <summary>

/// نام فیلد (با dot notation برای Nested Objects).

/// </summary>

public string PropertyName { get; set; }

/// <summary>

/// مقدار قبلی فیلد (به صورت JSON ذخیره می‌شود).

/// </summary>

public JsonElement OldValue { get; set; }

/// <summary>

/// مقدار جدید فیلد (به صورت JSON ذخیره می‌شود).

/// </summary>

public JsonElement NewValue { get; set; }

/// <summary>

/// شناسه کاربر عامل.

/// </summary>

public string ChangedBy { get; set; }

/// <summary>

/// IP Address کاربر.

/// </summary>

public string IpAddress { get; set; }

/// <summary>

/// User-Agent کاربر.

/// </summary>

public string UserAgent { get; set; }

/// <summary>

/// تاریخ و زمان تغییر.

/// </summary>

public DateTime ChangedAt { get; set; } = DateTime.UtcNow;

/// <summary>

/// دلیل تغییر (اختیاری).

/// </summary>

public string ChangeReason { get; set; }

/// <summary>

/// وضعیت حذف منطقی (Soft Delete).

/// </summary>

public bool IsDeleted { get; set; }

/// <summary>

/// زمان آخرین به‌روزرسانی.

/// </summary>

public DateTime? UpdatedAt { get; set; }

/// <summary>

/// متد برای سریالایز مقادیر به JsonElement.

/// </summary>

/// <param name="value">مقدار ورودی.</param>

/// <returns>JsonElement حاصل از سریالایز.</returns>

public static JsonElement SerializeToJsonElement(object value)

{

if (value == null)

return default;

try

{

var json = JsonSerializer.Serialize(value);

using var document = JsonDocument.Parse(json);

return document.RootElement.Clone();

}

catch

{

return default;

}

}

/// <summary>

/// متد برای دی‌سریالایز JsonElement به نوع دلخواه.

/// </summary>

/// <typeparam name="T">نوع خروجی.</typeparam>

/// <param name="jsonElement">JsonElement ورودی.</param>

/// <returns>مقدار دی‌سریالایز شده.</returns>

public static T DeserializeFromJsonElement<T>(JsonElement jsonElement)

{

if (jsonElement.ValueKind == JsonValueKind.Undefined || jsonElement.ValueKind == JsonValueKind.Null)

return default;

try

{

return JsonSerializer.Deserialize<T>(jsonElement.GetRawText());

}

catch

{

return default;

}

}

}

/// <summary>

/// انواع عملیات لاگ‌گیری.

/// </summary>

public enum AuditOperationType

{

/// <summary>

/// ایجاد رکورد جدید.

/// </summary>

Create,

/// <summary>

/// به‌روزرسانی رکورد موجود.

/// </summary>

Update,

/// <summary>

/// حذف رکورد.

/// </summary>

Delete

}

}

```

---

### \*\*توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس برای ذخیره تغییرات سیستمی (Audit Logs) طراحی شده است.

- تمامی اطلاعات مربوط به تغییرات (مانند نوع عملیات، نام موجودیت، مقادیر قدیمی و جدید، و اطلاعات کاربر) در این کلاس ذخیره می‌شوند.

2. \*\*فیلدهای کلاس:\*\*

- \*\*`Id`:\*\* شناسه یکتا برای هر لاگ.

- \*\*`OperationType`:\*\* نوع عملیات (Create, Update, Delete).

- \*\*`EntityName`:\*\* نام موجودیت (Entity) که تغییر در آن انجام شده است.

- \*\*`EntityId`:\*\* شناسه موجودیت.

- \*\*`PropertyName`:\*\* نام فیلد تغییر یافته.

- \*\*`OldValue` و `NewValue`:\*\* مقادیر قدیمی و جدید فیلد (به صورت JSON ذخیره می‌شوند).

- \*\*`ChangedBy`:\*\* شناسه کاربر عامل.

- \*\*`IpAddress`:\*\* آدرس IP کاربر.

- \*\*`UserAgent`:\*\* User-Agent کاربر.

- \*\*`ChangedAt`:\*\* تاریخ و زمان تغییر.

- \*\*`ChangeReason`:\*\* دلیل تغییر (اختیاری).

- \*\*`IsDeleted`:\*\* وضعیت حذف منطقی (Soft Delete).

- \*\*`UpdatedAt`:\*\* زمان آخرین به‌روزرسانی.

3. \*\*متدهای کمکی:\*\*

- \*\*`SerializeToJsonElement`:\*\* برای سریالایز مقادیر به `JsonElement`.

- \*\*`DeserializeFromJsonElement`:\*\* برای دی‌سریالایز `JsonElement` به نوع دلخواه.

4. \*\*بهبودها:\*\*

- \*\*مدیریت خطاهای سریالایز:\*\* در صورت بروز خطا در سریالایز یا دی‌سریالایز، مقدار پیش‌فرض (`default`) بازگردانده می‌شود.

- \*\*پشتیبانی از Soft Delete:\*\* فیلد `IsDeleted` برای مدیریت حذف منطقی اضافه شده است.

- \*\*زمان‌بندی دقیق:\*\* فیلدهای `ChangedAt` و `UpdatedAt` برای ثبت تاریخ و زمان دقیق تغییرات استفاده شده‌اند.

5. \*\*قابلیت توسعه:\*\*

- این کلاس طوری طراحی شده است که به راحتی قابل توسعه است. می‌توانید فیلدهای جدیدی مانند `CorrelationId` یا `TenantId` به آن اضافه کنید.

6. \*\*مثال استفاده:\*\*

#### \*\*1. ایجاد یک لاگ تغییرات:\*\*

```csharp

var auditEntry = new AuditEntry

{

OperationType = AuditOperationType.Update,

EntityName = "User",

EntityId = "12345",

PropertyName = "Email",

OldValue = AuditEntry.SerializeToJsonElement("old@example.com"),

NewValue = AuditEntry.SerializeToJsonElement("new@example.com"),

ChangedBy = "Admin",

IpAddress = "192.168.1.1",

UserAgent = "Mozilla/5.0",

ChangeReason = "Update user email"

};

// ذخیره در دیتابیس یا لاگ‌گیری

```

#### \*\*2. دی‌سریالایز مقادیر:\*\*

```csharp

var oldValue = AuditEntry.DeserializeFromJsonElement<string>(auditEntry.OldValue);

var newValue = AuditEntry.DeserializeFromJsonElement<string>(auditEntry.NewValue);

Console.WriteLine($"Old Value: {oldValue}, New Value: {newValue}");

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\* - کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدهای سریالایز و دی‌سریالایز به راحتی قابل تست هستند.

3. \*\*پشتیبانی از Soft Delete:\*\*

- فیلد `IsDeleted` برای مدیریت حذف منطقی اضافه شده است.

4. \*\*مدیریت خطاهای سریالایز:\*\*

- در صورت بروز خطا در سریالایز یا دی‌سریالایز، مقدار پیش‌فرض بازگردانده می‌شود.

5. \*\*قابلیت توسعه:\*\*

- امکان اضافه کردن فیلدهای جدید به راحتی وجود دارد.

در ادامه، کد کامل و بهینه‌شده برای `IAuditRepository` و پیاده‌سازی آن (`AuditRepository`) ارائه شده است. این کد شامل تمام جزئیات مورد نیاز، بهبودها و قابلیت‌های اضافی است که برای یک سیستم لاگ‌گیری حرفه‌ای لازم است. همه نکات مهم در طراحی و پیاده‌سازی در نظر گرفته شده‌اند.

\*\*IAuditRepository.cs\*\*

using System.Collections.Generic;

using System.Threading.Tasks;

namespace SharedKernel.Audit.Repositories

{

/// <summary>

/// Interface برای ذخیره لاگ‌ها در دیتابیس.

/// </summary>

public interface IAuditRepository

{

/// <summary>

/// ذخیره تغییرات در دیتابیس.

/// </summary>

/// <param name="changes">لیست تغییراتی که باید ذخیره شوند.</param>

/// <returns>Task برای عملیات ذخیره‌سازی.</returns>

Task SaveAsync(IEnumerable<Models.AuditChange> changes);

/// <summary>

/// دریافت تغییرات مرتبط با یک موجودیت خاص.

/// </summary>

/// <param name="entityName">نام موجودیت (Entity).</param>

/// <param name="entityId">شناسه موجودیت (Entity ID).</param>

/// <returns>لیستی از تغییرات مرتبط با موجودیت مشخص‌شده.</returns>

Task<IEnumerable<Models.AuditChange>> GetChangesByEntityAsync(string entityName, string entityId);

/// <summary>

/// دریافت تمام تغییرات در یک بازه زمانی مشخص.

/// </summary>

/// <param name="startDate">تاریخ شروع بازه.</param>

/// <param name="endDate">تاریخ پایان بازه.</param>

/// <returns>لیستی از تغییرات در بازه زمانی مشخص‌شده.</returns>

Task<IEnumerable<Models.AuditChange>> GetChangesByDateRangeAsync(DateTime startDate, DateTime endDate);

}

}

\*\*AuditRepository.cs\*\*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using SharedKernel.Audit.Models;

namespace SharedKernel.Audit.Repositories

{

/// <summary>

/// پیاده‌سازی Repository برای ذخیره لاگ‌ها در دیتابیس.

/// </summary>

public class AuditRepository : IAuditRepository

{

private readonly AuditDbContext \_context;

/// <summary>

/// سازنده کلاس.

/// </summary>

/// <param name="context">Instance از DbContext.</param>

public AuditRepository(AuditDbContext context)

{

\_context = context ?? throw new ArgumentNullException(nameof(context));

}

/// <summary>

/// ذخیره تغییرات در دیتابیس.

/// </summary>

/// <param name="changes">لیست تغییراتی که باید ذخیره شوند.</param>

/// <returns>Task برای عملیات ذخیره‌سازی.</returns>

public async Task SaveAsync(IEnumerable<AuditChange> changes)

{

if (changes == null || !changes.Any())

return;

try

{

await \_context.AuditChanges.AddRangeAsync(changes);

await \_context.SaveChangesAsync();

}

catch (Exception ex)

{

// ثبت خطای رخ‌داده در لاگ‌ها

Console.Error.WriteLine($"خطا در ذخیره لاگ‌ها: {ex.Message}");

throw new InvalidOperationException("خطا در ذخیره لاگ‌ها در دیتابیس.", ex);

}

}

/// <summary>

/// دریافت تغییرات مرتبط با یک موجودیت خاص.

/// </summary>

/// <param name="entityName">نام موجودیت (Entity).</param>

/// <param name="entityId">شناسه موجودیت (Entity ID).</param>

/// <returns>لیستی از تغییرات مرتبط با موجودیت مشخص‌شده.</returns>

public async Task<IEnumerable<AuditChange>> GetChangesByEntityAsync(string entityName, string entityId)

{

if (string.IsNullOrEmpty(entityName) || string.IsNullOrEmpty(entityId))

throw new ArgumentException("نام موجودیت و شناسه آن نباید خالی باشند.");

return await \_context.AuditChanges

.Where(ac => ac.EntityName == entityName && ac.EntityId == entityId)

.ToListAsync();

}

/// <summary>

/// دریافت تمام تغییرات در یک بازه زمانی مشخص.

/// </summary>

/// <param name="startDate">تاریخ شروع بازه.</param>

/// <param name="endDate">تاریخ پایان بازه.</param>

/// <returns>لیستی از تغییرات در بازه زمانی مشخص‌شده.</returns>

public async Task<IEnumerable<AuditChange>> GetChangesByDateRangeAsync(DateTime startDate, DateTime endDate)

{

if (startDate > endDate)

throw new ArgumentException("تاریخ شروع نباید بزرگ‌تر از تاریخ پایان باشد.");

return await \_context.AuditChanges

.Where(ac => ac.ChangedAt >= startDate && ac.ChangedAt <= endDate)

.ToListAsync();

}

}

}

\*\*AuditDbContext.cs\*\*

using Microsoft.EntityFrameworkCore;

using SharedKernel.Audit.Models;

namespace SharedKernel.Audit.Repositories

{

/// <summary>

/// DbContext برای مدیریت داده‌های لاگ‌گیری.

/// </summary>

public class AuditDbContext : DbContext

{

/// <summary>

/// DbSet برای لاگ‌های تغییرات.

/// </summary>

public DbSet<AuditChange> AuditChanges { get; set; }

/// <summary>

/// سازنده کلاس.

/// </summary>

/// <param name="options">تنظیمات DbContext.</param>

public AuditDbContext(DbContextOptions<AuditDbContext> options) : base(options)

{

}

/// <summary>

/// تنظیمات مدل.

/// </summary>

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// تنظیمات مربوط به جدول لاگ‌ها

modelBuilder.Entity<AuditChange>(entity =>

{

entity.HasKey(ac => ac.Id); // تنظیم کلید اصلی

entity.Property(ac => ac.EntityName).IsRequired().HasMaxLength(255); // نام موجودیت

entity.Property(ac => ac.EntityId).IsRequired().HasMaxLength(255); // شناسه موجودیت

entity.Property(ac => ac.PropertyName).HasMaxLength(255); // نام فیلد

entity.Property(ac => ac.ChangedBy).HasMaxLength(255); // کاربر عامل

entity.Property(ac => ac.IpAddress).HasMaxLength(50); // آدرس IP

entity.Property(ac => ac.UserAgent).HasMaxLength(500); // User-Agent

entity.Property(ac => ac.ChangedAt).HasDefaultValueSql("GETUTCDATE()"); // تاریخ تغییر

entity.Property(ac => ac.ChangeReason).HasMaxLength(1000); // دلیل تغییر

});

}

}

}

\*\*ServiceCollectionExtensions.cs\*\*

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.DependencyInjection;

using SharedKernel.Audit.Repositories;

namespace SharedKernel.Audit.Extensions

{

/// <summary>

/// Extension Methods برای ثبت سرویس‌های لاگ‌گیری.

/// </summary>

public static class ServiceCollectionExtensions

{

/// <summary>

/// افزودن سرویس‌های لاگ‌گیری به DI Container.

/// </summary>

/// <param name="services">کانتینر Dependency Injection.</param>

/// <param name="connectionString">رشته اتصال به دیتابیس.</param>

/// <returns>IServiceCollection پس از ثبت سرویس‌ها.</returns>

public static IServiceCollection AddAuditServices(this IServiceCollection services, string connectionString)

{

if (string.IsNullOrEmpty(connectionString))

throw new ArgumentException("رشته اتصال به دیتابیس نباید خالی باشد.");

// ثبت DbContext

services.AddDbContext<AuditDbContext>(options =>

options.UseSqlServer(connectionString));

// ثبت Repository

services.AddScoped<IAuditRepository, AuditRepository>();

return services;

}

}

}

\*\*توضیحات کد\*\*

1. \*\*Interface (`IAuditRepository`):\*\*

- شامل متدهایی برای ذخیره، بازیابی و جستجوی لاگ‌ها است.

- این Interface به عنوان قرارداد برای پیاده‌سازی‌های مختلف استفاده می‌شود.

2. \*\*پیاده‌سازی (`AuditRepository`):\*\*

- این کلاس مسئولیت تعامل مستقیم با دیتابیس را بر عهده دارد.

- شامل مدیریت خطاها و اعتبارسنجی ورودی‌ها است.

3. \*\*DbContext (`AuditDbContext`):\*\*

- شامل تنظیمات مدل داده‌ای و محدودیت‌های مربوط به جداول دیتابیس است.

- از `OnModelCreating` برای تنظیم دقیق جدول لاگ‌ها استفاده شده است.

4. \*\*Extension Method (`AddAuditServices`):\*\*

- برای تسهیل ثبت سرویس‌ها در DI Container طراحی شده است.

- شامل تنظیمات اتصال به دیتابیس و ثبت Repository است.

5. \*\*بهبودها:\*\*

- \*\*مدیریت خطا:\*\* در صورت بروز خطا، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

- \*\*اعتبارسنجی ورودی‌ها:\*\* بررسی صحت ورودی‌ها قبل از اجرای عملیات.

- \*\*پشتیبانی از Query Optimization:\*\* استفاده از `IQueryable` برای بهینه‌سازی جستجوها.

- \*\*تنظیمات مدل:\*\* استفاده از `OnModelCreating` برای تنظیم دقیق جداول دیتابیس.

6. \*\*قابلیت توسعه:\*\*

- این طراحی به راحتی قابل توسعه است. می‌توانید متدهای جدیدی مانند `DeleteLogsAsync` یا `PurgeOldDataAsync` به آن اضافه کنید.

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدها به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از Query Optimization:\*\*

- استفاده از `IQueryable` برای بهینه‌سازی جستجوها و کاهش بار دیتابیس.

4. \*\*مدیریت خطاهای سیستمی:\*\*

- در صورت بروز خطا، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

5. \*\*قابلیت گسترش:\*\*

- امکان اضافه کردن متدهای جدید به راحتی وجود دارد.

در ادامه، کد کامل و بهینه‌شده برای کلاس `AuditSaveChangesInterceptor` ارائه شده است. این کد شامل تمامی بهبودها، جزئیات و قابلیت‌های لازم برای لاگ‌گیری خودکار تغییرات در عملیات `SaveChanges` است. همه نکات مهم در طراحی و پیاده‌سازی در نظر گرفته شده‌اند.

\*\*AuditSaveChangesInterceptor.cs\*\*

using Microsoft.EntityFrameworkCore.Diagnostics;

using Microsoft.EntityFrameworkCore;

using SharedKernel.Audit.Helpers;

using SharedKernel.Audit.Models;

using SharedKernel.Audit.Repositories;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

namespace SharedKernel.Audit

{

/// <summary>

/// Interceptor برای لاگ‌گیری خودکار تغییرات در SaveChanges.

/// </summary>

public class AuditSaveChangesInterceptor : SaveChangesInterceptor

{

private readonly IAuditRepository \_auditRepository;

/// <summary>

/// سازنده کلاس.

/// </summary>

/// <param name="auditRepository">Instance از IAuditRepository برای ذخیره لاگ‌ها.</param>

public AuditSaveChangesInterceptor(IAuditRepository auditRepository)

{

\_auditRepository = auditRepository ?? throw new ArgumentNullException(nameof(auditRepository));

}

/// <summary>

/// متد برای لاگ‌گیری خودکار تغییرات قبل از ذخیره‌سازی در دیتابیس.

/// </summary>

/// <param name="eventData">اطلاعات رویداد مرتبط با SaveChanges.</param>

/// <param name="result">نتیجه میانی عملیات SaveChanges.</param>

/// <param name="cancellationToken">توکن لغو عملیات.</param>

/// <returns>InterceptionResult حاوی نتیجه عملیات.</returns>

public override async ValueTask<InterceptionResult<int>> SavingChangesAsync(

DbContextEventData eventData,

InterceptionResult<int> result,

CancellationToken cancellationToken = default)

{

// بررسی وجود Context

if (eventData.Context == null)

return await base.SavingChangesAsync(eventData, result, cancellationToken);

// استخراج تغییرات از ChangeTracker

var changes = new List<AuditChange>();

foreach (var entry in eventData.Context.ChangeTracker.Entries())

{

// دریافت نوع موجودیت و شناسه آن

var entityType = entry.Entity.GetType();

var entityId = GetEntityId(entry) ?? Guid.NewGuid().ToString();

// تعیین نوع عملیات و استخراج تغییرات

switch (entry.State)

{

case EntityState.Added:

changes.AddRange(AuditTrailHelper.GetAuditChanges(

null,

entry.Entity,

"System",

entityId,

AuditOperationType.Create));

break;

case EntityState.Modified:

changes.AddRange(AuditTrailHelper.GetAuditChanges(

entry.OriginalValues.ToObject(),

entry.Entity,

"System",

entityId,

AuditOperationType.Update));

break;

case EntityState.Deleted:

changes.AddRange(AuditTrailHelper.GetAuditChanges(

entry.Entity,

null,

"System",

entityId,

AuditOperationType.Delete));

break;

}

}

// ذخیره تغییرات در دیتابیس

if (changes.Any())

{

try

{

await \_auditRepository.SaveAsync(changes);

}

catch (Exception ex)

{

// ثبت خطای رخ‌داده در لاگ‌ها

Console.Error.WriteLine($"خطا در ذخیره لاگ‌ها: {ex.Message}");

throw new InvalidOperationException("خطا در ذخیره لاگ‌ها در دیتابیس.", ex);

}

}

return await base.SavingChangesAsync(eventData, result, cancellationToken);

}

/// <summary>

/// دریافت شناسه موجودیت (Entity ID).

/// </summary>

/// <param name="entry">ورودی ChangeTracker.</param>

/// <returns>شناسه موجودیت به صورت رشته.</returns>

private string GetEntityId(EntityEntry entry)

{

try

{

// بررسی وجود پراپرتی "Id" یا "ID"

var idProperty = entry.Properties.FirstOrDefault(p => p.Metadata.Name.Equals("Id", StringComparison.OrdinalIgnoreCase));

return idProperty?.CurrentValue?.ToString();

}

catch

{

// در صورت بروز خطا، شناسه پیش‌فرض برگردانده می‌شود

return null;

}

}

}

}

\*\*توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس به عنوان یک `Interceptor` برای Entity Framework Core عمل می‌کند و تغییرات اعمال‌شده در دیتابیس را به صورت خودکار لاگ می‌کند.

- از `SaveChangesInterceptor` برای دریافت تغییرات قبل از ذخیره‌سازی استفاده می‌شود.

2. \*\*فیلدها و متدها:\*\*

- \*\*`\_auditRepository`:\*\* مخزن (Repository) مورد استفاده برای ذخیره لاگ‌ها.

- \*\*`SavingChangesAsync`:\*\* متد اصلی برای لاگ‌گیری تغییرات.

- \*\*`GetEntityId`:\*\* متد کمکی برای دریافت شناسه موجودیت (Entity ID).

3. \*\*بهبودها:\*\*

- \*\*مدیریت خطاهای سیستمی:\*\*

- در صورت بروز خطا در ذخیره لاگ‌ها، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

- \*\*بررسی وجود پراپرتی "Id":\*\*

- اگر موجودیت شناسه‌ای نداشته باشد، یک شناسه پیش‌فرض (`Guid.NewGuid()`) تولید می‌شود.

- \*\*استفاده از `ValueTask`:\*\*

- برای بهینه‌سازی عملیات‌های ناهمزمان، از `ValueTask` استفاده شده است.

- \*\*پشتیبانی از انواع مختلف شناسه:\*\*

- متد `GetEntityId` به صورت پویا شناسه موجودیت را استخراج می‌کند.

4. \*\*مثال استفاده:\*\*

1. تنظیمات در `DbContext`:\*\*

```csharp

public class AppDbContext : DbContext

{

private readonly IAuditRepository \_auditRepository;

public AppDbContext(DbContextOptions<AppDbContext> options, IAuditRepository auditRepository)

: base(options)

{

\_auditRepository = auditRepository;

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

base.OnConfiguring(optionsBuilder);

// افزودن Interceptor

optionsBuilder.AddInterceptors(new AuditSaveChangesInterceptor(\_auditRepository));

}

}

\*\*2. تست لاگ‌گیری:\*\*

```csharp

var entity = new User { Name = "John Doe" };

dbContext.Users.Add(entity);

await dbContext.SaveChangesAsync(); // لاگ‌گیری خودکار تغییرات

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدها به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از خطاهای سیستمی:\*\*

- در صورت بروز خطا، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

4. \*\*مدیریت تغییرات:\*\*

- تمامی تغییرات اعمال‌شده در دیتابیس به صورت خودکار لاگ می‌شوند.

5. \*\*قابلیت توسعه:\*\*

- امکان اضافه کردن قابلیت‌های جدید (مانند فیلتر کردن لاگ‌ها) به راحتی وجود دارد.

در ادامه، کد کامل و بهینه‌شده برای کلاس `LoggingBehavior` ارائه شده است. این کلاس شامل تمامی بهبودها و جزئیات مورد نیاز برای لاگ‌گیری دقیق، فیلتر کردن داده‌های حساس، اندازه‌گیری مدت زمان اجرای عملیات، و استفاده از `LogScope` است. همه نکات مهم در طراحی و پیاده‌سازی در نظر گرفته شده‌اند.

\*\*LoggingBehavior.cs\*\*

```csharp

using System;

using System.Diagnostics;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

using MediatR;

using Microsoft.Extensions.Logging;

namespace SharedKernel.Audit.Behaviors

{

/// <summary>

/// Behavior برای لاگ‌گیری خودکار در Pipelines.

/// </summary>

/// <typeparam name="TRequest">نوع درخواست.</typeparam>

/// <typeparam name="TResponse">نوع پاسخ.</typeparam>

public class LoggingBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>

where TRequest : IRequest<TResponse>

{

private readonly ILogger<LoggingBehavior<TRequest, TResponse>> \_logger;

/// <summary>

/// سازنده کلاس.

/// </summary>

/// <param name="logger">Instance از ILogger برای لاگ‌گیری.</param>

public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger)

{

\_logger = logger ?? throw new ArgumentNullException(nameof(logger));

}

/// <summary>

/// متد اصلی برای اجرای لاگ‌گیری قبل و بعد از اجرای عملیات.

/// </summary>

/// <param name="request">درخواست ورودی.</param>

/// <param name="next">تابع بعدی در Pipeline.</param>

/// <param name="cancellationToken">توکن لغو عملیات.</param>

/// <returns>نتیجه عملیات.</returns>

public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)

{

// دریافت نام درخواست

var requestName = typeof(TRequest).Name;

// فیلتر کردن پارامترهای حساس

var sensitiveFields = new[] { "Password", "Token", "NationalId" };

var filteredParams = request?.GetType().GetProperties()

.Select(p =>

{

var value = sensitiveFields.Contains(p.Name) ? "\*\*\*" : p.GetValue(request)?.ToString();

return $"{p.Name}: {value}";

}) ?? Array.Empty<string>();

var requestParameters = string.Join(", ", filteredParams);

// شروع StopWatch برای اندازه‌گیری زمان اجرای عملیات

var stopwatch = Stopwatch.StartNew();

// ایجاد LogScope برای مدیریت Context

using (\_logger.BeginScope("RequestId: {RequestId}, RequestType: {RequestType}", Guid.NewGuid(), requestName))

{

try

{

// لاگ‌گیری شروع عملیات

\_logger.LogInformation("Handling {RequestName} with parameters: {RequestParameters}", requestName, requestParameters);

// اجرای عملیات اصلی

var response = await next();

// توقف StopWatch و لاگ‌گیری پایان موفقیت‌آمیز عملیات

stopwatch.Stop();

\_logger.LogInformation("Successfully handled {RequestName} in {ElapsedMilliseconds}ms. Response: {@Response}",

requestName, stopwatch.ElapsedMilliseconds, response);

return response;

}

catch (Exception ex)

{

// توقف StopWatch و لاگ‌گیری خطاهای رخ‌داده

stopwatch.Stop();

\_logger.LogError(ex, "Error occurred while handling {RequestName} in {ElapsedMilliseconds}ms. Error: {ErrorMessage}",

requestName, stopwatch.ElapsedMilliseconds, ex.Message);

throw;

}

}

}

}

}

\*\*توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس به عنوان یک `Pipeline Behavior` در MediatR عمل می‌کند و تضمین می‌کند که هر درخواست و پاسخی که از طریق MediatR ارسال می‌شود، به صورت خودکار لاگ شود.

- این لاگ‌ها شامل اطلاعات درخواست، پارامترها، پاسخ، مدت زمان اجرا و خطاهای رخ‌داده می‌شوند.

2. \*\*فیلدها و متدها:\*\*

- \*\*`\_logger`:\*\* Instance از `ILogger` برای ثبت لاگ‌ها.

- \*\*`Handle`:\*\* متد اصلی برای اجرای لاگ‌گیری قبل و بعد از اجرای عملیات.

3. \*\*بهبودها:\*\*

- \*\*مدیریت خطاهای سیستمی:\*\*

- در صورت بروز خطا، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

- \*\*پشتیبانی از پارامترهای درخواست:\*\*

- پارامترهای درخواست به صورت خودکار استخراج و لاگ می‌شوند.

- \*\*فیلتر کردن داده‌های حساس:\*\*

- اطلاعات حساس مانند پسورد، توکن و شماره ملی به صورت خودکار فیلتر می‌شوند.

- \*\*استفاده از `Stopwatch` برای اندازه‌گیری زمان اجرا:\*\*

- مدت زمان اجرای هر عملیات به صورت دقیق اندازه‌گیری و لاگ می‌شود.

- \*\*استفاده از `LogScope` برای مدیریت Context:\*\*

- اطلاعات Context مانند شناسه درخواست و نوع درخواست به صورت جداگانه لاگ می‌شوند.

4. \*\*مثال استفاده:\*\*

\*\*1. تنظیمات در `Program.cs`:\*\*

```csharp

builder.Services.AddMediatR(cfg =>

{

cfg.RegisterServicesFromAssembly(typeof(Program).Assembly);

cfg.AddBehavior(typeof(IPipelineBehavior<,>), typeof(LoggingBehavior<,>));

});

\*\*2. تست لاگ‌گیری:\*\*

```csharp

public class GetUserQuery : IRequest<UserDto>

{

public int UserId { get; set; }

public string Password { get; set; } // فیلد حساس

}

public class GetUserQueryHandler : IRequestHandler<GetUserQuery, UserDto>

{

public Task<UserDto> Handle(GetUserQuery request, CancellationToken cancellationToken)

{

return Task.FromResult(new UserDto { Id = request.UserId, Name = "John Doe" });

}

}

// ارسال درخواست

var user = await mediator.Send(new GetUserQuery { UserId = 1, Password = "Secret@123" });

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدها به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از خطاهای سیستمی:\*\*

- در صورت بروز خطا، پیام‌های مناسبی به لاگ‌ها اضافه می‌شود.

4. \*\*مدیریت تغییرات:\*\*

- تمامی درخواست‌ها و پاسخ‌ها به صورت خودکار لاگ می‌شوند.

5. \*\*قابلیت توسعه:\*\*

- امکان اضافه کردن قابلیت‌های جدید (مانند فیلتر کردن لاگ‌ها) به راحتی وجود دارد.

6. \*\*حفاظت از اطلاعات حساس:\*\*

- اطلاعات حساس مانند پسورد و توکن به صورت خودکار فیلتر می‌شوند.

7. \*\*اندازه‌گیری عملکرد:\*\*

- مدت زمان اجرای هر عملیات به صورت دقیق اندازه‌گیری و لاگ می‌شود.

در ادامه، کد کامل و بهینه‌شده برای `BaseEntity` با در نظر گرفتن تمامی پیشنهادات و نکات مطرح‌شده ارائه شده است. این کلاس شامل فیلدهای عمومی، متدهای کمکی، و قابلیت‌های پیشرفته مانند `Audit Log`، `Concurrency Control`، و انعطاف‌پذیری بیشتر با استفاده از اینترفیس است.

---

### \*\*BaseEntity.cs\*\*

```csharp

using System;

using System.ComponentModel.DataAnnotations;

namespace SharedKernel.Common

{

/// <summary>

/// اینترفیس عمومی برای تمام موجودیت‌ها.

/// </summary>

public interface IEntity

{

/// <summary>

/// شناسه یکتا برای هر موجودیت.

/// </summary>

Guid Id { get; set; }

}

/// <summary>

/// کلاس پایه برای تمام موجودیت‌های دیتابیس.

/// </summary>

public abstract class BaseEntity : IEntity

{

/// <summary>

/// شناسه یکتا برای هر موجودیت.

/// </summary>

public virtual Guid Id { get; set; } = Guid.NewGuid();

/// <summary>

/// تاریخ و زمان ایجاد موجودیت.

/// </summary>

public virtual DateTime CreatedAt { get; set; } = DateTime.UtcNow;

/// <summary>

/// تاریخ و زمان آخرین به‌روزرسانی موجودیت.

/// </summary>

public virtual DateTime? UpdatedAt { get; set; }

/// <summary>

/// وضعیت حذف منطقی (Soft Delete).

/// </summary>

public virtual bool IsDeleted { get; set; }

/// <summary>

/// شناسه کاربری که این موجودیت را ایجاد کرده.

/// </summary>

public virtual Guid? CreatedBy { get; set; }

/// <summary>

/// شناسه کاربری که آخرین بار این موجودیت را ویرایش کرده.

/// </summary>

public virtual Guid? UpdatedBy { get; set; }

/// <summary>

/// نسخه‌بندی برای کنترل همزمانی (Concurrency Control).

/// </summary>

[Timestamp]

public virtual byte[] RowVersion { get; set; }

/// <summary>

/// متد برای به‌روزرسانی تاریخ آخرین تغییر.

/// </summary>

/// <param name="userId">شناسه کاربری که تغییر را انجام داده.</param>

public void UpdateTimestamp(Guid? userId = null)

{

UpdatedAt = DateTime.UtcNow;

UpdatedBy = userId;

}

/// <summary>

/// متد برای نشانه‌گذاری موجودیت به عنوان حذف‌شده.

/// </summary>

/// <param name="userId">شناسه کاربری که حذف را انجام داده.</param>

public void MarkAsDeleted(Guid? userId = null)

{

IsDeleted = true;

UpdateTimestamp(userId);

}

/// <summary>

/// متد برای نشانه‌گذاری موجودیت به عنوان بازیابی‌شده.

/// </summary>

/// <param name="userId">شناسه کاربری که بازیابی را انجام داده.</param>

public void MarkAsRestored(Guid? userId = null)

{

IsDeleted = false;

UpdateTimestamp(userId);

}

}

}

\*\*توضیحات کد\*\*

1. \*\*هدف از کلاس:\*\*

- این کلاس به عنوان یک کلاس پایه برای تمام موجودیت‌های دیتابیس عمل می‌کند.

- شامل فیلدهای عمومی مانند `Id`, `CreatedAt`, `UpdatedAt`, `IsDeleted`, `CreatedBy`, `UpdatedBy`, و `RowVersion` است.

2. \*\*فیلدها:\*\*

- \*\*`Id`:\*\* شناسه یکتا برای هر موجودیت. به صورت پیش‌فرض، یک `Guid` جدید تولید می‌شود.

- \*\*`CreatedAt`:\*\* تاریخ و زمان ایجاد موجودیت. به صورت خودکار در زمان ایجاد موجودیت تنظیم می‌شود.

- \*\*`UpdatedAt`:\*\* تاریخ و زمان آخرین به‌روزرسانی موجودیت. این فیلد اختیاری است و فقط زمانی تنظیم می‌شود که موجودیت به‌روزرسانی شود.

- \*\*`IsDeleted`:\*\* وضعیت حذف منطقی. اگر این فیلد `true` باشد، یعنی موجودیت به صورت منطقی حذف شده است.

- \*\*`CreatedBy`:\*\* شناسه کاربری که موجودیت را ایجاد کرده است.

- \*\*`UpdatedBy`:\*\* شناسه کاربری که آخرین بار موجودیت را به‌روزرسانی کرده است.

- \*\*`RowVersion`:\*\* نسخه‌بندی برای کنترل همزمانی (Concurrency Control).

3. \*\*متدهای کمکی:\*\*

- \*\*`UpdateTimestamp`:\*\* برای به‌روزرسانی تاریخ آخرین تغییر و شناسه کاربری.

- \*\*`MarkAsDeleted`:\*\* برای نشانه‌گذاری موجودیت به عنوان حذف‌شده.

- \*\*`MarkAsRestored`:\*\* برای نشانه‌گذاری موجودیت به عنوان بازیابی‌شده.

4. \*\*بهبودها:\*\*

- \*\*مدیریت زمان‌بندی دقیق:\*\* از `DateTime.UtcNow` برای ذخیره تاریخ و زمان دقیق استفاده شده است.

- \*\*پشتیبانی از Soft Delete:\*\* فیلد `IsDeleted` برای مدیریت حذف منطقی اضافه شده است.

- \*\*کنترل همزمانی:\*\* از `[Timestamp]` برای کنترل همزمانی استفاده شده است.

- \*\*استفاده از اینترفیس:\*\* کلاس `BaseEntity` از اینترفیس `IEntity` ارث‌بری می‌کند تا انعطاف‌پذیری بیشتری داشته باشد.

5. \*\*مثال استفاده:\*\*

\*\*1. ایجاد یک موجودیت جدید:\*\*

```csharp

public class UserEntity : BaseEntity

{

public string Name { get; set; }

public string Email { get; set; }

}

var user = new UserEntity

{

Name = "John Doe",

Email = "john.doe@example.com",

CreatedBy = Guid.NewGuid() // شناسه کاربری که ایجاد کرده

};

// تاریخ ایجاد به صورت خودکار تنظیم می‌شود

Console.WriteLine($"Created At: {user.CreatedAt}");

\*\*2. به‌روزرسانی موجودیت:\*\*

```csharp

user.Name = "Jane Doe";

user.UpdateTimestamp(Guid.NewGuid()); // شناسه کاربری که به‌روزرسانی کرده

Console.WriteLine($"Updated At: {user.UpdatedAt}");

\*\*3. حذف منطقی موجودیت:\*\*

```csharp

user.MarkAsDeleted(Guid.NewGuid()); // شناسه کاربری که حذف کرده

Console.WriteLine($"Is Deleted: {user.IsDeleted}");

Console.WriteLine($"Updated At After Deletion: {user.UpdatedAt}");

\*\*4. بازیابی موجودیت:\*\*

```csharp

user.MarkAsRestored(Guid.NewGuid()); // شناسه کاربری که بازیابی کرده

Console.WriteLine($"Is Deleted: {user.IsDeleted}");

Console.WriteLine($"Updated At After Restoration: {user.UpdatedAt}");

\*\*مزایای این طراحی\*\*

1. \*\*خوانایی بالا:\*\*

- کد تمیز و خوانا است و از کامنت‌های فارسی دقیق استفاده شده است.

2. \*\*قابلیت تست‌پذیری:\*\*

- متدها به راحتی قابل تست هستند و نیازی به تکرار منطق در نقاط مختلف ندارید.

3. \*\*پشتیبانی از Soft Delete:\*\*

- فیلد `IsDeleted` برای مدیریت حذف منطقی اضافه شده است.

4. \*\*مدیریت زمان‌بندی دقیق:\*\*

- از `DateTime.UtcNow` برای ذخیره تاریخ و زمان دقیق استفاده شده است.

5. \*\*کنترل همزمانی:\*\*

- از `[Timestamp]` برای کنترل همزمانی استفاده شده است.

6. \*\*قابلیت توسعه:\*\*

- امکان اضافه کردن فیلدهای جدید به راحتی وجود دارد.

7. \*\*انعطاف‌پذیری:\*\*

- استفاده از اینترفیس `IEntity` برای انعطاف‌پذیری بیشتر.

برای پیاده‌سازی کامل و بهینه‌شده `ValueObject`، در ادامه یک نسخه جامع و کامل ارائه می‌شود. این نسخه شامل تمامی جزئیات لازم برای پیاده‌سازی Value Objects در یک سیستم Domain-Driven Design (DDD) است. این شامل کلاس پایه (`ValueObject`)، نمونه‌های Value Object (مانند `EmailAddress`, `PhoneNumber`, و `GeoCoordinate`)، اعتبارسنجی، مقایسه عمیق، و مدیریت خطا است.

\*\*1. ValueObject.cs (کلاس پایه)\*\*

```csharp

using System;

using System.Collections.Generic;

using System.Linq;

namespace SharedKernel.Common.ValueObjects

{

/// <summary>

/// کلاس پایه برای تمام Value Objects.

/// </summary>

public abstract class ValueObject : IEquatable<ValueObject>

{

/// <summary>

/// مقایسه دو Value Object بر اساس مقادیر آنها.

/// </summary>

public bool Equals(ValueObject other)

{

if (other is null)

return false;

return GetEqualityComponents().SequenceEqual(other.GetEqualityComponents());

}

/// <summary>

/// مقایسه دو Value Object بر اساس مقادیر آنها.

/// </summary>

public override bool Equals(object obj)

=> obj is ValueObject other && Equals(other);

/// <summary>

/// محاسبه HashCode بر اساس مقادیر Value Object.

/// </summary>

public override int GetHashCode()

{

return GetEqualityComponents()

.Select(x => x?.GetHashCode() ?? 0)

.Aggregate((x, y) => x ^ y);

}

/// <summary>

/// عملگر تساوی برای مقایسه دو Value Object.

/// </summary>

public static bool operator ==(ValueObject a, ValueObject b)

{

if (ReferenceEquals(a, b))

return true;

if (a is null || b is null)

return false;

return a.Equals(b);

}

/// <summary>

/// عملگر نامساوی برای مقایسه دو Value Object.

/// </summary>

public static bool operator !=(ValueObject a, ValueObject b)

{

return !(a == b);

}

/// <summary>

/// این متد باید در کلاس‌های فرزند پیاده‌سازی شود.

/// </summary>

protected abstract IEnumerable<object> GetEqualityComponents();

}

}

\*\*2. EmailAddress.cs (Value Object برای ایمیل)\*\*

```csharp

using System;

using System.Text.RegularExpressions;

namespace SharedKernel.Common.ValueObjects

{

/// <summary>

/// Value Object برای نمایش آدرس ایمیل.

/// </summary>

public sealed class EmailAddress : ValueObject

{

/// <summary>

/// مقدار ایمیل.

/// </summary>

public string Value { get; }

/// <summary>

/// سازنده برای اعتبارسنجی و مقداردهی اولیه.

/// </summary>

/// <param name="value">مقدار ایمیل.</param>

public EmailAddress(string value)

{

if (!Regex.IsMatch(value, @"^[^@\s]+@[^@\s]+\.[^@\s]+$"))

throw new ArgumentException("Invalid email format");

Value = value;

}

/// <summary>

/// پیاده‌سازی GetEqualityComponents برای مقایسه.

/// </summary>

protected override IEnumerable<object> GetEqualityComponents()

{

yield return Value;

}

/// <summary>

/// نمایش رشته‌ای ایمیل.

/// </summary>

public override string ToString() => Value;

}

}

\*\*3. PhoneNumber.cs (Value Object برای شماره تلفن)\*\*

```csharp

using System;

using System.Text.RegularExpressions;

namespace SharedKernel.Common.ValueObjects

{

/// <summary>

/// Value Object برای نمایش شماره تلفن.

/// </summary>

public sealed class PhoneNumber : ValueObject

{

/// <summary>

/// مقدار شماره تلفن.

/// </summary>

public string Value { get; }

/// <summary>

/// سازنده برای اعتبارسنجی و مقداردهی اولیه.

/// </summary>

/// <param name="value">مقدار شماره تلفن.</param>

public PhoneNumber(string value)

{

if (!Regex.IsMatch(value, @"^\+?[0-9]{10,15}$"))

throw new ArgumentException("Invalid phone number format");

Value = value;

}

/// <summary>

/// پیاده‌سازی GetEqualityComponents برای مقایسه.

/// </summary>

protected override IEnumerable<object> GetEqualityComponents()

{

yield return Value;

}

/// <summary>

/// نمایش رشته‌ای شماره تلفن.

/// </summary>

public override string ToString() => Value;

}

}

\*\*4. GeoCoordinate.cs (Value Object برای مختصات جغرافیایی)\*\*

```csharp

using System;

namespace SharedKernel.Common.ValueObjects

{

/// <summary>

/// Value Object برای نمایش مختصات جغرافیایی.

/// </summary>

public sealed class GeoCoordinate : ValueObject

{

/// <summary>

/// عرض جغرافیایی.

/// </summary>

public double Latitude { get; }

/// <summary>

/// طول جغرافیایی.

/// </summary>

public double Longitude { get; }

/// <summary>

/// سازنده برای اعتبارسنجی و مقداردهی اولیه.

/// </summary>

/// <param name="latitude">عرض جغرافیایی.</param>

/// <param name="longitude">طول جغرافیایی.</param>

public GeoCoordinate(double latitude, double longitude)

{

if (latitude < -90 || latitude > 90 || longitude < -180 || longitude > 180)

throw new ArgumentException("Invalid geographic coordinates");

Latitude = latitude;

Longitude = longitude;

}

/// <summary>

/// پیاده‌سازی GetEqualityComponents برای مقایسه.

/// </summary>

protected override IEnumerable<object> GetEqualityComponents()

{

yield return Latitude;

yield return Longitude;

}

/// <summary>

/// نمایش رشته‌ای مختصات جغرافیایی.

/// </summary>

public override string ToString() => $"Latitude: {Latitude}, Longitude: {Longitude}";

}

}

\*\*5. CustomException.cs (استثناء سفارشی)\*\*

using System;

namespace SharedKernel.Common.Exceptions

{

/// <summary>

/// استثناء سفارشی برای مدیریت خطاها.

/// </summary>

public class CustomException : Exception

{

/// <summary>

/// کد خطا.

/// </summary>

public int ErrorCode { get; }

/// <summary>

/// سازنده با پیام و کد خطا.

/// </summary>

public CustomException(string message, int errorCode = 400)

: base(message)

{

ErrorCode = errorCode;

}

}

}

\*\*توضیحات کلی\*\*

1. \*\*Immutable Design:\*\*

- همه Value Objects به‌صورت Immutable طراحی شده‌اند. این به این معناست که پس از ایجاد یک Value Object، مقادیر آن قابل تغییر نیستند.

2. \*\*اعتبارسنجی:\*\*

- هر Value Object شامل اعتبارسنجی‌های سفارشی برای اطمینان از صحت داده‌ها است.

3. \*\*مقایسه عمیق:\*\*

- از `GetEqualityComponents` برای مقایسه دقیق Value Objects استفاده شده است.

4. \*\*مدیریت خطا:\*\*

- از `CustomException` برای مدیریت خطاهای اعتبارسنجی استفاده شده است.

5. \*\*قابلیت توسعه:\*\*

- امکان اضافه کردن Value Objects جدید به راحتی وجود دارد.

\*\*مثال استفاده\*\*

```csharp

var email1 = new EmailAddress("john.doe@example.com");

var email2 = new EmailAddress("john.doe@example.com");

Console.WriteLine(email1 == email2); // True

var phone1 = new PhoneNumber("+1234567890");

var phone2 = new PhoneNumber("+1234567890");

Console.WriteLine(phone1 == phone2); // True

var coordinate1 = new GeoCoordinate(35.6895, 51.3890);

var coordinate2 = new GeoCoordinate(35.6895, 51.3890);

Console.WriteLine(coordinate1 == coordinate2); // True

\*\*جمع‌بندی\*\*

این پیاده‌سازی کامل و بهینه‌شده از `ValueObject` شامل:

1. \*\*کلاس پایه (`ValueObject`):\*\*

- شامل متدهای مقایسه (`Equals`), محاسبه `HashCode`, و عملگرهای `==` و `!=`.

2. \*\*نمونه‌های Value Object:\*\*

- `EmailAddress`: برای نمایش آدرس ایمیل.

- `PhoneNumber`: برای نمایش شماره تلفن.

- `GeoCoordinate`: برای نمایش مختصات جغرافیایی.

3. \*\*اعتبارسنجی:\*\*

- هر Value Object شامل اعتبارسنجی‌های سفارشی است.

4. \*\*مدیریت خطا:\*\*

- از `CustomException` برای مدیریت خطاهای اعتبارسنجی استفاده شده است.

در ادامه، کد کامل و بهینه‌شده برای دو کلاس `Result.cs` و `PaginatedResult.cs` ارائه شده است. این کلاس‌ها برای مدیریت نتایج عملیات‌ها (مانند موفقیت یا شکست) و همچنین مدیریت نتایج پیاده‌سازی شده‌اند. این کلاس‌ها شامل جزئیات کامل و بهبودهای لازم برای استفاده در سراسر پروژه هستند.

\*\*1. Result.cs\*\*

using System;

namespace SharedKernel.Common

{

/// <summary>

/// کلاس عمومی برای مدیریت نتایج عملیات‌ها.

/// </summary>

/// <typeparam name="T">نوع داده‌ای که نتیجه برمی‌گرداند.</typeparam>

public class Result<T>

{

/// <summary>

/// وضعیت موفقیت عملیات.

/// </summary>

public bool IsSuccess { get; }

/// <summary>

/// پیام مرتبط با نتیجه.

/// </summary>

public string Message { get; }

/// <summary>

/// داده‌های برگشتی در صورت موفقیت.

/// </summary>

public T Data { get; }

/// <summary>

/// خطا (در صورت وجود).

/// </summary>

public Exception Exception { get; }

/// <summary>

/// سازنده برای نتیجه موفق.

/// </summary>

/// <param name="data">داده‌های برگشتی.</param>

/// <param name="message">پیام موفقیت.</param>

private Result(T data, string message)

{

IsSuccess = true;

Data = data;

Message = message;

}

/// <summary>

/// سازنده برای نتیجه ناموفق.

/// </summary>

/// <param name="message">پیام خطا.</param>

/// <param name="exception">خطای رخ‌داده (اختیاری).</param>

private Result(string message, Exception exception = null)

{

IsSuccess = false;

Message = message;

Exception = exception;

}

/// <summary>

/// ایجاد نتیجه موفق.

/// </summary>

/// <param name="data">داده‌های برگشتی.</param>

/// <param name="message">پیام موفقیت.</param>

public static Result<T> Success(T data, string message = "Operation completed successfully.")

{

return new Result<T>(data, message);

}

/// <summary>

/// ایجاد نتیجه ناموفق.

/// </summary>

/// <param name="message">پیام خطا.</param>

/// <param name="exception">خطای رخ‌داده (اختیاری).</param>

public static Result<T> Failure(string message, Exception exception = null)

{

return new Result<T>(message, exception);

}

/// <summary>

/// نمایش رشته‌ای نتیجه.

/// </summary>

public override string ToString()

{

return IsSuccess

? $"Success: {Message}"

: $"Failure: {Message}{(Exception != null ? $" - Exception: {Exception.Message}" : "")}";

}

}

}

\*\*2. PaginatedResult.cs\*\*

using System.Collections.Generic;

namespace SharedKernel.Common

{

/// <summary>

/// کلاس عمومی برای مدیریت نتایج پیاده‌سازی‌شده.

/// </summary>

/// <typeparam name="T">نوع داده‌ای که نتیجه برمی‌گرداند.</typeparam>

public class PaginatedResult<T>

{

/// <summary>

/// وضعیت موفقیت عملیات.

/// </summary>

public bool IsSuccess { get; }

/// <summary>

/// پیام مرتبط با نتیجه.

/// </summary>

public string Message { get; }

/// <summary>

/// تعداد کل آیتم‌ها.

/// </summary>

public int TotalCount { get; }

/// <summary>

/// تعداد صفحات.

/// </summary>

public int TotalPages { get; }

/// <summary>

/// صفحه فعلی.

/// </summary>

public int CurrentPage { get; }

/// <summary>

/// تعداد آیتم‌های هر صفحه.

/// </summary>

public int PageSize { get; }

/// <summary>

/// داده‌های برگشتی.

/// </summary>

public IEnumerable<T> Data { get; }

/// <summary>

/// خطا (در صورت وجود).

/// </summary>

public Exception Exception { get; }

/// <summary>

/// سازنده برای نتیجه موفق.

/// </summary>

/// <param name="data">داده‌های برگشتی.</param>

/// <param name="totalCount">تعداد کل آیتم‌ها.</param>

/// <param name="currentPage">صفحه فعلی.</param>

/// <param name="pageSize">تعداد آیتم‌های هر صفحه.</param>

/// <param name="message">پیام موفقیت.</param>

private PaginatedResult(IEnumerable<T> data, int totalCount, int currentPage, int pageSize, string message)

{

IsSuccess = true;

Data = data;

TotalCount = totalCount;

CurrentPage = currentPage;

PageSize = pageSize;

TotalPages = (int)Math.Ceiling(totalCount / (double)pageSize);

Message = message;

}

/// <summary>

/// سازنده برای نتیجه ناموفق.

/// </summary>

/// <param name="message">پیام خطا.</param>

/// <param name="exception">خطای رخ‌داده (اختیاری).</param>

private PaginatedResult(string message, Exception exception = null)

{

IsSuccess = false;

Message = message;

Exception = exception;

}

/// <summary>

/// ایجاد نتیجه موفق برای داده‌های پیاده‌سازی‌شده.

/// </summary>

/// <param name="data">داده‌های برگشتی.</param>

/// <param name="totalCount">تعداد کل آیتم‌ها.</param>

/// <param name="currentPage">صفحه فعلی.</param>

/// <param name="pageSize">تعداد آیتم‌های هر صفحه.</param>

/// <param name="message">پیام موفقیت.</param>

public static PaginatedResult<T> Success(IEnumerable<T> data, int totalCount, int currentPage, int pageSize, string message = "Operation completed successfully.")

{

return new PaginatedResult<T>(data, totalCount, currentPage, pageSize, message);

}

/// <summary>

/// ایجاد نتیجه ناموفق.

/// </summary>

/// <param name="message">پیام خطا.</param>

/// <param name="exception">خطای رخ‌داده (اختیاری).</param>

public static PaginatedResult<T> Failure(string message, Exception exception = null)

{

return new PaginatedResult<T>(message, exception);

}

/// <summary>

/// نمایش رشته‌ای نتیجه.

/// </summary>

public override string ToString()

{

return IsSuccess

? $"Success: {Message} - Page {CurrentPage}/{TotalPages}, Total Items: {TotalCount}"

: $"Failure: {Message}{(Exception != null ? $" - Exception: {Exception.Message}" : "")}";

}

}

}

\*\*توضیحات کلی\*\*

\*\*1. Result.cs\*\*

- \*\*هدف:\*\* این کلاس برای مدیریت نتایج عملیات‌ها (مانند موفقیت یا شکست) استفاده می‌شود.

- \*\*ویژگی‌ها:\*\*

- `IsSuccess`: مشخص می‌کند که آیا عملیات موفق بوده است یا خیر.

- `Message`: پیام مرتبط با نتیجه.

- `Data`: داده‌های برگشتی در صورت موفقیت.

- `Exception`: خطای رخ‌داده در صورت وجود.

- \*\*متدها:\*\*

- `Success`: برای ایجاد نتیجه موفق.

- `Failure`: برای ایجاد نتیجه ناموفق.

- `ToString`: برای نمایش رشته‌ای نتیجه.

\*\*2. PaginatedResult.cs\*\*

- \*\*هدف:\*\* این کلاس برای مدیریت نتایج پیاده‌سازی‌شده استفاده می‌شود.

- \*\*ویژگی‌ها:\*\*

- `IsSuccess`: مشخص می‌کند که آیا عملیات موفق بوده است یا خیر.

- `Message`: پیام مرتبط با نتیجه.

- `TotalCount`: تعداد کل آیتم‌ها.

- `TotalPages`: تعداد صفحات.

- `CurrentPage`: صفحه فعلی.

- `PageSize`: تعداد آیتم‌های هر صفحه.

- `Data`: داده‌های برگشتی.

- `Exception`: خطای رخ‌داده در صورت وجود.

- \*\*متدها:\*\*

- `Success`: برای ایجاد نتیجه موفق برای داده‌های پیاده‌سازی‌شده.

- `Failure`: برای ایجاد نتیجه ناموفق.

- `ToString`: برای نمایش رشته‌ای نتیجه.

\*\*مثال استفاده\*\*

\*\*1. استفاده از `Result<T>`:\*\*

```csharp

var result = Result<string>.Success("Data loaded successfully.", "Sample Data");

if (result.IsSuccess)

{

Console.WriteLine($"Success: {result.Data}");

}

else

{

Console.WriteLine($"Failure: {result.Message}");

}

\*\*2. استفاده از `PaginatedResult<T>`:\*\*

var data = new List<string> { "Item1", "Item2", "Item3" };

var paginatedResult = PaginatedResult<string>.Success(data, 100, 1, 10, "Data fetched successfully.");

if (paginatedResult.IsSuccess)

{

Console.WriteLine($"Page {paginatedResult.CurrentPage}/{paginatedResult.TotalPages}, Total Items: {paginatedResult.TotalCount}");

}

else

{

Console.WriteLine($"Failure: {paginatedResult.Message}");

}

\*\*جمع‌بندی\*\*

این پیاده‌سازی شامل تمام جزئیات لازم برای مدیریت نتایج عملیات‌ها و داده‌های پیاده‌سازی‌شده است.

در ادامه، کد کامل و بهینه‌شده برای فایل `AppSettings.cs` در پوشه `Config` ارائه شده است. این فایل شامل تمامی تنظیمات مورد نیاز برنامه است که از فایل `appsettings.json` خوانده می‌شوند. این تنظیمات شامل اطلاعات اتصال به دیتابیس، تنظیمات Redis، Kafka، و سایر پارامترهای سیستمی است. همچنین، بهینه‌سازی‌هایی مانند اعتبارسنجی و مدیریت خطاهای مربوط به تنظیمات نیز اعمال شده است.

---

### \*\*1. AppSettings.cs (کلاس تنظیمات)\*\*

```csharp

using System.ComponentModel.DataAnnotations;

namespace SharedKernel.Config

{

/// <summary>

/// کلاس اصلی برای مدیریت تنظیمات برنامه.

/// </summary>

public class AppSettings

{

/// <summary>

/// تنظیمات اتصال به دیتابیس.

/// </summary>

[Required(ErrorMessage = "Database connection settings are required.")]

public ConnectionStrings ConnectionStrings { get; set; }

/// <summary>

/// تنظیمات Redis.

/// </summary>

public RedisSettings Redis { get; set; }

/// <summary>

/// تنظیمات Kafka.

/// </summary>

public KafkaSettings Kafka { get; set; }

/// <summary>

/// تنظیمات لاگ‌زنی.

/// </summary>

public LoggingSettings Logging { get; set; }

/// <summary>

/// تنظیمات عمومی برنامه.

/// </summary>

public GeneralSettings General { get; set; }

}

/// <summary>

/// تنظیمات اتصال به دیتابیس.

/// </summary>

public class ConnectionStrings

{

/// <summary>

/// رشته اتصال به دیتابیس اصلی.

/// </summary>

[Required(ErrorMessage = "DefaultConnection string is required.")]

public string DefaultConnection { get; set; }

}

/// <summary>

/// تنظیمات Redis.

/// </summary>

public class RedisSettings

{

/// <summary>

/// آدرس اتصال به Redis.

/// </summary>

[Required(ErrorMessage = "Redis connection string is required.")]

public string ConnectionString { get; set; }

}

/// <summary>

/// تنظیمات Kafka.

/// </summary>

public class KafkaSettings

{

/// <summary>

/// آدرس سرورهای Kafka.

/// </summary>

[Required(ErrorMessage = "Kafka bootstrap servers are required.")]

public string BootstrapServers { get; set; }

}

/// <summary>

/// تنظیمات لاگ‌زنی.

/// </summary>

public class LoggingSettings

{

/// <summary>

/// سطح لاگ‌زنی پیش‌فرض.

/// </summary>

public string LogLevel { get; set; } = "Information";

/// <summary>

/// سطح لاگ‌زنی برای ASP.NET Core.

/// </summary>

public string MicrosoftAspNetCoreLogLevel { get; set; } = "Warning";

}

/// <summary>

/// تنظیمات عمومی برنامه.

/// </summary>

public class GeneralSettings

{

/// <summary>

/// حداکثر تعداد تلاش برای ارسال پیام‌ها.

/// </summary>

public int MaxRetryAttempts { get; set; } = 3;

/// <summary>

/// زمان تاخیر بین تلاش‌ها (به میلی‌ثانیه).

/// </summary>

public int RetryDelayMilliseconds { get; set; } = 1000;

}

}

\*\*2. appsettings.json (فایل تنظیمات JSON)\*\*

```json

{

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=AppDb;Trusted\_Connection=True;"

},

"Redis": {

"ConnectionString": "localhost:6379"

},

"Kafka": {

"BootstrapServers": "localhost:9092"

},

"Logging": {

"LogLevel": "Information",

"Microsoft.AspNetCore": "Warning"

},

"General": {

"MaxRetryAttempts": 3,

"RetryDelayMilliseconds": 1000

}

}

\*\*3. AppSettingsService.cs (سرویس خواندن تنظیمات)\*\*

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using SharedKernel.Config;

namespace SharedKernel.Extensions

{

/// <summary>

/// سرویس‌های مرتبط با تنظیمات برنامه.

/// </summary>

public static class AppSettingsService

{

/// <summary>

/// افزودن تنظیمات برنامه به سرویس‌های Dependency Injection.

/// </summary>

public static IServiceCollection AddAppSettings(this IServiceCollection services, IConfiguration configuration)

{

// اعتبارسنجی تنظیمات

var appSettings = configuration.Get<AppSettings>();

if (appSettings == null)

{

throw new InvalidOperationException("AppSettings is not configured properly.");

}

// اضافه کردن تنظیمات به DI Container

services.AddSingleton(appSettings);

return services;

}

}

}

\*\*4. Program.cs (استفاده از تنظیمات)\*\*

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using SharedKernel.Extensions;

var builder = WebApplication.CreateBuilder(args);

// بارگذاری تنظیمات از appsettings.json

builder.Services.AddAppSettings(builder.Configuration);

var app = builder.Build();

// استفاده از تنظیمات در برنامه

var appSettings = app.Services.GetService<AppSettings>();

if (appSettings != null)

{

Console.WriteLine($"Database Connection: {appSettings.ConnectionStrings.DefaultConnection}");

Console.WriteLine($"Redis Connection: {appSettings.Redis.ConnectionString}");

Console.WriteLine($"Kafka Servers: {appSettings.Kafka.BootstrapServers}");

}

app.Run();

\*\*توضیحات کلی\*\*

1. \*\*اعتبارسنجی:\*\*

- از Attribute `Required` برای اطمینان از وجود تنظیمات ضروری استفاده شده است.

- اگر هر یک از تنظیمات ضروری موجود نباشد، خطای مناسبی ایجاد می‌شود.

2. \*\*مدیریت تنظیمات:\*\*

- تنظیمات در قالب کلاس‌های جداگانه (`ConnectionStrings`, `RedisSettings`, `KafkaSettings`, و غیره) سازمان‌دهی شده‌اند.

- این رویکرد به مدیریت و توسعه تنظیمات کمک می‌کند.

3. \*\*Dependency Injection:\*\*

- تنظیمات به‌صورت Singleton به سرویس‌های Dependency Injection اضافه شده‌اند.

- این امکان را فراهم می‌کند که تنظیمات در هر جایی از برنامه قابل دسترسی باشند.

4. \*\*مدیریت خطاهای تنظیمات:\*\*

- اگر تنظیمات به‌درستی بارگذاری نشوند، خطای مناسبی ایجاد می‌شود.

5. \*\*پیکربندی JSON:\*\*

- تنظیمات از فایل `appsettings.json` خوانده می‌شوند و به کلاس‌های C# تبدیل می‌شوند.

\*\*مثال استفاده\*\*

\*\*1. دسترسی به تنظیمات در کنترلر:\*\*

```csharp

using Microsoft.AspNetCore.Mvc;

using SharedKernel.Config;

[ApiController]

[Route("api/[controller]")]

public class SettingsController : ControllerBase

{

private readonly AppSettings \_appSettings;

public SettingsController(AppSettings appSettings)

{

\_appSettings = appSettings;

}

[HttpGet]

public IActionResult Get()

{

return Ok(new

{

Database = \_appSettings.ConnectionStrings.DefaultConnection,

Redis = \_appSettings.Redis.ConnectionString,

Kafka = \_appSettings.Kafka.BootstrapServers

});

}

}

\*\*2. تست تنظیمات:\*\*

```csharp

var appSettings = new AppSettings

{

ConnectionStrings = new ConnectionStrings

{

DefaultConnection = "Server=localhost;Database=TestDb;Trusted\_Connection=True;"

},

Redis = new RedisSettings

{

ConnectionString = "localhost:6379"

},

Kafka = new KafkaSettings

{

BootstrapServers = "localhost:9092"

}

};

if (string.IsNullOrEmpty(appSettings.ConnectionStrings.DefaultConnection))

{

throw new Exception("DefaultConnection is missing.");

}

\*\*جمع‌بندی\*\*

این پیاده‌سازی شامل تمام جزئیات لازم برای مدیریت تنظیمات برنامه است:

1. \*\*کلاس‌های تنظیمات:\*\* شامل کلاس‌های جداگانه برای هر بخش از تنظیمات.

2. \*\*اعتبارسنجی:\*\* اطمینان از وجود تنظیمات ضروری.

3. \*\*Dependency Injection:\*\* تسهیل دسترسی به تنظیمات در سراسر برنامه.

4. \*\*مدیریت خطاهای تنظیمات:\*\* ایجاد خطاهای مناسب در صورت فقدان تنظیمات.

در ادامه، کد کامل و بهینه‌شده برای فایل `AppConstants.cs` در پوشه `Constants` ارائه شده است. این فایل شامل ثابت‌هایی است که در سراسر پروژه استفاده می‌شوند. این ثابت‌ها می‌توانند شامل نام‌های پیش‌فرض اتصالات دیتابیس، تنظیمات کلیدی سیستم، مسیرهای پیش‌فرض، یا هر نوع مقدار ثابتی باشند که در بخش‌های مختلف برنامه مورد نیاز است.

---

### \*\*1. AppConstants.cs (کلاس ثابت‌ها)\*\*

```csharp

namespace SharedKernel.Constants

{

/// <summary>

/// کلاس ثابت‌ها برای استفاده در سراسر پروژه.

/// </summary>

public static class AppConstants

{

/// <summary>

/// نام پیش‌فرض رشته اتصال به دیتابیس.

/// </summary>

public const string DefaultConnectionStringName = "DefaultConnection";

/// <summary>

/// مسیر پیش‌فرض لاگ‌ها.

/// </summary>

public const string DefaultLogPath = "Logs/app-log.txt";

/// <summary>

/// حداکثر تعداد تلاش برای ارسال پیام‌ها.

/// </summary>

public const int MaxRetryAttempts = 3;

/// <summary>

/// زمان تاخیر بین تلاش‌ها (به میلی‌ثانیه).

/// </summary>

public const int RetryDelayMilliseconds = 1000;

/// <summary>

/// مدت زمان اعتبار توکن JWT (به دقیقه).

/// </summary>

public const int JwtTokenExpirationMinutes = 60;

/// <summary>

/// نام کلید خاص برای ذخیره‌سازی توکن‌ها در Redis.

/// </summary>

public const string RedisTokenKeyPrefix = "AuthToken:";

/// <summary>

/// نام کلید خاص برای ذخیره‌سازی Session در Redis.

/// </summary>

public const string RedisSessionKeyPrefix = "UserSession:";

/// <summary>

/// آدرس پیش‌فرض سرور Kafka.

/// </summary>

public const string KafkaBootstrapServers = "localhost:9092";

/// <summary>

/// نام Topic پیش‌فرض در Kafka.

/// </summary>

public const string KafkaDefaultTopic = "default-topic";

/// <summary>

/// حداکثر اندازه فایل‌های آپلودی (به بایت).

/// </summary>

public const long MaxUploadFileSizeBytes = 5 \* 1024 \* 1024; // 5MB

/// <summary>

/// لیست فرمت‌های فایل مجاز برای آپلود.

/// </summary>

public static readonly string[] AllowedFileExtensions = { ".jpg", ".jpeg", ".png", ".pdf" };

/// <summary>

/// نام کلید خاص برای ذخیره‌سازی تنظیمات کاربر در Cache.

/// </summary>

public const string UserSettingsCacheKey = "UserSettings:";

/// <summary>

/// مدت زمان اعتبار داده‌ها در Cache (به دقیقه).

/// </summary>

public const int CacheExpirationMinutes = 30;

/// <summary>

/// نام سرویس‌های داخلی (Microservices).

/// </summary>

public static class ServiceNames

{

public const string AuthService = "AuthService";

public const string DeviceService = "DeviceService";

public const string NotificationService = "NotificationService";

public const string BillingService = "BillingService";

}

/// <summary>

/// تنظیمات مربوط به API Gateway.

/// </summary>

public static class ApiGateway

{

public const string BaseUrl = "https://api.example.com";

public const string HealthCheckEndpoint = "/health";

public const string StatusEndpoint = "/status";

}

}

}

```

---

### \*\*2. توضیحات جامع\*\*

#### \*\*1. ثابت‌های دیتابیس\*\*

- `DefaultConnectionStringName`: نام پیش‌فرض رشته اتصال به دیتابیس که معمولاً در فایل `appsettings.json` تعریف می‌شود.

- `RedisTokenKeyPrefix` و `RedisSessionKeyPrefix`: پیش‌وند‌هایی برای ذخیره‌سازی توکن‌ها و Session‌ها در Redis.

#### \*\*2. تنظیمات لاگ‌زنی\*\*

- `DefaultLogPath`: مسیر پیش‌فرض فایل لاگ‌ها.

#### \*\*3. تنظیمات ارسال پیام‌ها\*\*

- `MaxRetryAttempts`: حداکثر تعداد تلاش برای ارسال پیام‌ها.

- `RetryDelayMilliseconds`: زمان تاخیر بین تلاش‌ها.

#### \*\*4. تنظیمات امنیتی\*\*

- `JwtTokenExpirationMinutes`: مدت زمان اعتبار توکن JWT.

- `AllowedFileExtensions`: لیست فرمت‌های فایل مجاز برای آپلود.

#### \*\*5. تنظیمات Kafka\*\*

- `KafkaBootstrapServers`: آدرس سرورهای Kafka.

- `KafkaDefaultTopic`: نام Topic پیش‌فرض در Kafka.

#### \*\*6. تنظیمات Cache\*\*

- `UserSettingsCacheKey`: کلید خاص برای ذخیره‌سازی تنظیمات کاربر در Cache.

- `CacheExpirationMinutes`: مدت زمان اعتبار داده‌ها در Cache.

#### \*\*7. تنظیمات Microservices\*\*

- `ServiceNames`: شامل نام‌های سرویس‌های داخلی (Microservices) مانند `AuthService`, `DeviceService`, و غیره.

#### \*\*8. تنظیمات API Gateway\*\*

- `ApiGateway`: شامل URL پایه و نقاط پایانی مربوط به API Gateway.

---

### \*\*3. مثال استفاده\*\*

#### \*\*1. استفاده از ثابت‌ها در کد:\*\*

```csharp

using SharedKernel.Constants;

public class ExampleUsage

{

public void LogToFile(string message)

{

var logPath = AppConstants.DefaultLogPath;

System.IO.File.AppendAllText(logPath, message + Environment.NewLine);

}

public void ConnectToDatabase()

{

var connectionStringName = AppConstants.DefaultConnectionStringName;

Console.WriteLine($"Connecting to database using connection string: {connectionName}");

}

public void PublishToKafka(string message)

{

var bootstrapServers = AppConstants.KafkaBootstrapServers;

var topic = AppConstants.KafkaDefaultTopic;

Console.WriteLine($"Publishing to Kafka - Servers: {bootstrapServers}, Topic: {topic}");

}

}

```

#### \*\*2. استفاده در تنظیمات Dependency Injection:\*\*

```csharp

using Microsoft.Extensions.DependencyInjection;

using SharedKernel.Constants;

public static class ServiceCollectionExtensions

{

public static IServiceCollection AddCustomServices(this IServiceCollection services)

{

services.AddSingleton(options =>

{

return new RedisOptions

{

ConnectionString = AppConstants.RedisTokenKeyPrefix,

ExpirationMinutes = AppConstants.CacheExpirationMinutes

};

});

return services;

}

}

\*\*4. مزایای استفاده از `AppConstants`\*\*

1. \*\*مدیریت مرکزی:\*\* تمامی ثابت‌ها در یک محل متمرکز قرار دارند و مدیریت آن‌ها آسان است.

2. \*\*جلوگیری از خطاهای تایپی:\*\* با استفاده از ثابت‌ها، از خطاهای تایپی در کد جلوگیری می‌شود.

3. \*\*قابلیت توسعه:\*\* اضافه کردن یا تغییر ثابت‌ها به‌راحتی امکان‌پذیر است.

4. \*\*استفاده مجدد:\*\* ثابت‌ها در هر بخش از پروژه قابل استفاده هستند.

در ادامه، کد کامل و بهینه‌شده برای فایل‌های `Roles.cs` و `ClaimTypes.cs` در پوشه‌های مربوطه ارائه شده است. این فایل‌ها شامل تعریف نقش‌ها (Roles) و نوع‌های ادعاهای (Claims) استفاده‌شده در سیستم هستند. این تعریف‌ها به‌صورت مرکزی مدیریت می‌شوند تا از یکپارچگی و سازگاری در سراسر پروژه اطمینان حاصل شود.

\*\*1. Roles.cs\*\*

namespace SharedKernel.Enums

{

/// <summary>

/// تعریف نقش‌های سیستم.

/// </summary>

public static class Roles

{

/// <summary>

/// نقش مدیر سیستم.

/// </summary>

public const string Admin = "Admin";

/// <summary>

/// نقش کاربر عادی.

/// </summary>

public const string User = "User";

/// <summary>

/// نقش مدیر بخش.

/// </summary>

public const string Manager = "Manager";

/// <summary>

/// نقش مشتری.

/// </summary>

public const string Customer = "Customer";

/// <summary>

/// نقش توسعه‌دهنده.

/// </summary>

public const string Developer = "Developer";

/// <summary>

/// نقش ناظر.

/// </summary>

public const string Auditor = "Auditor";

/// <summary>

/// لیست تمام نقش‌ها.

/// </summary>

public static readonly string[] AllRoles = { Admin, User, Manager, Customer, Developer, Auditor };

}

}

\*\*2. ClaimTypes.cs\*\*

using System.Security.Claims;

namespace SharedKernel.Constants

{

/// <summary>

/// تعریف نوع‌های ادعایی (Claims) مورد استفاده در سیستم.

/// </summary>

public static class ClaimTypes

{

/// <summary>

/// ادعای شناسه کاربر.

/// </summary>

public const string UserId = "UserId";

/// <summary>

/// ادعای نام کاربری.

/// </summary>

public const string UserName = "UserName";

/// <summary>

/// ادعای ایمیل کاربر.

/// </summary>

public const string Email = "Email";

/// <summary>

/// ادعای نقش کاربر.

/// </summary>

public const string Role = "Role";

/// <summary>

/// ادعای شماره تلفن کاربر.

/// </summary>

public const string PhoneNumber = "PhoneNumber";

/// <summary>

/// ادعای وضعیت کاربر (فعال/غیرفعال).

/// </summary>

public const string UserStatus = "UserStatus";

/// <summary>

/// ادعای دسترسی‌های خاص کاربر.

/// </summary>

public const string Permissions = "Permissions";

/// <summary>

/// ادعای زمان آخرین ورود کاربر.

/// </summary>

public const string LastLoginDate = "LastLoginDate";

/// <summary>

/// ادعای شناسه جلسه کاربر.

/// </summary>

public const string SessionId = "SessionId";

/// <summary>

/// ادعای IP کاربر.

/// </summary>

public const string IpAddress = "IpAddress";

/// <summary>

/// ادعای شناسه سازمانی کاربر.

/// </summary>

public const string OrganizationId = "OrganizationId";

/// <summary>

/// ادعای شناسه دستگاه.

/// </summary>

public const string DeviceId = "DeviceId";

}

}

\*\*3. توضیحات جامع\*\*

\*\*1. Roles.cs\*\*

- \*\*هدف:\*\* این فایل شامل تعریف نقش‌های سیستم است که برای مدیریت دسترسی‌ها و اختیارات کاربران استفاده می‌شود.

- \*\*ویژگی‌ها:\*\*

- `Admin`: نقش مدیر سیستم.

- `User`: نقش کاربر عادی.

- `Manager`: نقش مدیر بخش.

- `Customer`: نقش مشتری.

- `Developer`: نقش توسعه‌دهنده.

- `Auditor`: نقش ناظر.

- `AllRoles`: لیست تمام نقش‌ها برای استفاده در محل‌های مختلف.

\*\*2. ClaimTypes.cs\*\*

- \*\*هدف:\*\* این فایل شامل تعریف نوع‌های ادعایی (Claims) است که در سیستم احراز هویت و مدیریت دسترسی استفاده می‌شوند.

- \*\*ویژگی‌ها:\*\*

- `UserId`: شناسه منحصر به فرد کاربر.

- `UserName`: نام کاربری.

- `Email`: آدرس ایمیل کاربر.

- `Role`: نقش کاربر.

- `PhoneNumber`: شماره تلفن کاربر.

- `UserStatus`: وضعیت کاربر (فعال/غیرفعال).

- `Permissions`: دسترسی‌های خاص کاربر.

- `LastLoginDate`: زمان آخرین ورود کاربر.

- `SessionId`: شناسه جلسه کاربر.

- `IpAddress`: آدرس IP کاربر.

- `OrganizationId`: شناسه سازمانی کاربر.

- `DeviceId`: شناسه دستگاه.

4. مثال استفاده\*\*

1. استفاده از نقش‌ها در سیستم احراز هویت:\*\*

using Microsoft.AspNetCore.Authorization;

using SharedKernel.Enums;

[Authorize(Roles = Roles.Admin)]

public IActionResult AdminDashboard()

{

return View();

}

[Authorize(Roles = Roles.User)]

public IActionResult UserProfile()

{

return View();

}

\*\*2. استفاده از ادعاهای کاربری:\*\*

using System.Security.Claims;

using SharedKernel.Constants;

public string GetUserId(ClaimsPrincipal user)

{

return user.FindFirstValue(ClaimTypes.UserId);

}

public string GetUserRole(ClaimsPrincipal user)

{

return user.FindFirstValue(ClaimTypes.Role);

}

public bool IsUserActive(ClaimsPrincipal user)

{

var status = user.FindFirstValue(ClaimTypes.UserStatus);

return status == "Active";

}

\*\*3. اضافه کردن ادعاهای کاربری به توکن JWT:\*\*

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using SharedKernel.Constants;

var claims = new List<Claim>

{

new Claim(ClaimTypes.UserId, userId),

new Claim(ClaimTypes.UserName, userName),

new Claim(ClaimTypes.Email, email),

new Claim(ClaimTypes.Role, role),

new Claim(ClaimTypes.PhoneNumber, phoneNumber),

new Claim(ClaimTypes.UserStatus, "Active"),

new Claim(ClaimTypes.SessionId, sessionId),

new Claim(ClaimTypes.IpAddress, ipAddress)

};

var token = new JwtSecurityToken(

issuer: "your-issuer",

audience: "your-audience",

claims: claims,

expires: DateTime.UtcNow.AddHours(1),

signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);

\*\*5. مزایای استفاده از `Roles` و `ClaimTypes`\*\*

1. \*\*مدیریت مرکزی:\*\* تمامی نقش‌ها و ادعاهای کاربری در یک محل متمرکز قرار دارند.

2. \*\*جلوگیری از خطاهای تایپی:\*\* با استفاده از ثابت‌ها، از خطاهای تایپی در کد جلوگیری می‌شود.

3. \*\*قابلیت توسعه:\*\* اضافه کردن یا تغییر نقش‌ها و ادعاهای کاربری به‌راحتی امکان‌پذیر است.

4. \*\*استفاده مجدد:\*\* نقش‌ها و ادعاهای کاربری در هر بخش از پروژه قابل استفاده هستند.

در ادامه، کد کامل و بهینه‌شده برای فایل `NotificationType.cs` در پوشه `Enums` ارائه شده است. این فایل شامل تعریف انواع اعلان‌ها (Notifications) است که در سراسر پروژه استفاده می‌شود. این تعریف‌ها به‌صورت مرکزی مدیریت می‌شوند تا از یکپارچگی و سازگاری در سراسر برنامه اطمینان حاصل شود.

\*\*1. NotificationType.cs\*\*

namespace SharedKernel.Enums

{

/// <summary>

/// تعریف انواع اعلان‌ها (Notifications).

/// </summary>

public enum NotificationType

{

/// <summary>

/// اطلاعیه عمومی.

/// </summary>

Info,

/// <summary>

/// هشدار.

/// </summary>

Warning,

/// <summary>

/// اعلام خطر.

/// </summary>

Alert,

/// <summary>

/// نتیجه اجرای دستور.

/// </summary>

CommandResult,

/// <summary>

/// اعلان موفقیت.

/// </summary>

Success,

/// <summary>

/// اعلان خطا.

/// </summary>

Error,

/// <summary>

/// اعلان سیستمی.

/// </summary>

SystemMessage

}

}

\*\*2. توضیحات جامع\*\*

\*\*1. هدف:\*\*

- \*\*مدیریت انواع اعلان‌ها:\*\* این Enum برای دسته‌بندی انواع اعلان‌ها استفاده می‌شود. این دسته‌بندی به مدیریت و نمایش اعلان‌ها کمک می‌کند.

- \*\*استفاده در سراسر پروژه:\*\* این Enum در بخش‌های مختلفی از پروژه مانند کنترلرها، سرویس‌ها، مخازن، و مدل‌ها استفاده می‌شود.

\*\*2. مقادیر تعریف‌شده:\*\*

- \*\*Info:\*\* اطلاعیه عمومی که معمولاً برای ارسال اطلاعات غیرضروری استفاده می‌شود.

- \*\*Warning:\*\* هشداری که به کاربر اطلاع می‌دهد که چیزی ممکن است مشکل‌ساز شود.

- \*\*Alert:\*\* اعلام خطر برای وضعیت‌های بحرانی.

- \*\*CommandResult:\*\* نتیجه اجرای یک دستور یا عملیات.

- \*\*Success:\*\* اعلان موفقیت برای عملیات‌هایی که با موفقیت انجام شده‌اند.

- \*\*Error:\*\* اعلان خطا برای عملیات‌هایی که با شکست مواجه شده‌اند.

- \*\*SystemMessage:\*\* اعلان‌های سیستمی که معمولاً برای مدیریت داخلی استفاده می‌شوند.

\*\*3. ویژگی‌ها:\*\*

- \*\*قابلیت توسعه:\*\* اگر نیاز به اضافه کردن نوع جدیدی از اعلان وجود داشته باشد، به‌راحتی می‌توان آن را به این Enum اضافه کرد.

- \*\*مرکزیت:\*\* تمامی انواع اعلان‌ها در یک محل متمرکز قرار دارند و مدیریت آن‌ها آسان است.

\*\*3. مثال استفاده\*\*

\*\*1. استفاده در مدل‌ها:\*\*

\*\*NotificationDto.cs\*\*

using SharedKernel.Enums;

namespace NotificationService.Models

{

/// <summary>

/// مدل انتقال داده‌ها برای اعلان‌ها.

/// </summary>

public class NotificationDto

{

/// <summary>

/// شناسه اعلان.

/// </summary>

public Guid Id { get; set; }

/// <summary>

/// شناسه کاربر.

/// </summary>

public string UserId { get; set; }

/// <summary>

/// عنوان اعلان.

/// </summary>

public string Title { get; set; }

/// <summary>

/// متن اعلان.

/// </summary>

public string Message { get; set; }

/// <summary>

/// نوع اعلان.

/// </summary>

public NotificationType Type { get; set; }

/// <summary>

/// آیا اعلان خوانده شده است؟

/// </summary>

public bool IsRead { get; set; }

/// <summary>

/// تاریخ ایجاد اعلان.

/// </summary>

public DateTime CreatedAt { get; set; }

}

}

\*\*2. استفاده در سرویس‌ها:\*\*

\*\*NotificationService.cs\*\*

```csharp

using SharedKernel.Enums;

using NotificationService.Models;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace NotificationService.Services

{

/// <summary>

/// سرویس مدیریت اعلان‌ها.

/// </summary>

public class NotificationService : INotificationService

{

/// <summary>

/// ارسال اعلان.

/// </summary>

public async Task SendNotificationAsync(NotificationDto notificationDto)

{

if (notificationDto.Type == NotificationType.Alert)

{

// ارسال اعلان خطر به کاربر

Console.WriteLine($"Alert sent to user {notificationDto.UserId}: {notificationDto.Message}");

}

else if (notificationDto.Type == NotificationType.Success)

{

// ارسال اعلان موفقیت

Console.WriteLine($"Success notification sent to user {notificationDto.UserId}: {notificationDto.Message}");

}

}

/// <summary>

/// دریافت اعلان‌های یک کاربر.

/// </summary>

public async Task<List<NotificationDto>> GetUserNotificationsAsync(string userId)

{

// در اینجا اعلان‌های کاربر از دیتابیس یا Cache دریافت می‌شود

return new List<NotificationDto>();

}

}

}

\*\*3. استفاده در کنترلرها:\*\*

\*\*NotificationsController.cs\*\*

```csharp

using Microsoft.AspNetCore.Mvc;

using SharedKernel.Enums;

using NotificationService.Models;

using NotificationService.Services;

using System.Threading.Tasks;

namespace NotificationService.Controllers

{

/// <summary>

/// کنترلر مدیریت اعلان‌ها.

/// </summary>

[ApiController]

[Route("api/[controller]")]

public class NotificationsController : ControllerBase

{

private readonly INotificationService \_notificationService;

public NotificationsController(INotificationService notificationService)

{

\_notificationService = notificationService;

}

/// <summary>

/// ارسال اعلان جدید.

/// </summary>

[HttpPost]

public async Task<IActionResult> SendNotification([FromBody] NotificationDto notificationDto)

{

await \_notificationService.SendNotificationAsync(notificationDto);

return Ok("Notification sent successfully.");

}

/// <summary>

/// دریافت اعلان‌های یک کاربر.

/// </summary>

[HttpGet("{userId}")]

public async Task<IActionResult> GetUserNotifications(string userId)

{

var notifications = await \_notificationService.GetUserNotificationsAsync(userId);

return Ok(notifications);

}

}

}

\*\*4. مزایای استفاده از `NotificationType`\*\*

1. \*\*مدیریت مرکزی:\*\* تمامی انواع اعلان‌ها در یک محل متمرکز قرار دارند.

2. \*\*جلوگیری از خطاهای تایپی:\*\* با استفاده از Enum، از خطاهای تایپی در کد جلوگیری می‌شود.

3. \*\*قابلیت توسعه:\*\* اضافه کردن یا تغییر انواع اعلان‌ها به‌راحتی امکان‌پذیر است.

4. \*\*استفاده مجدد:\*\* انواع اعلان‌ها در هر بخش از پروژه قابل استفاده هستند.

در ادامه، کد کامل و بهینه‌شده برای فایل `CommandType.cs` ارائه شده است. این فایل شامل یک Enum (انواع شمارشی) است که انواع دستورات قابل ارسال به دستگاه‌ها را تعریف می‌کند. این Enum به‌صورت مرکزی مدیریت می‌شود تا از یکپارچگی و سازگاری در سراسر پروژه اطمینان حاصل شود.

---

### \*\*1. CommandType.cs\*\*

```csharp

namespace SharedKernel.Enums

{

/// <summary>

/// تعریف انواع دستورات قابل ارسال به دستگاه‌ها.

/// </summary>

public enum CommandType

{

/// <summary>

/// راه‌اندازی مجدد دستگاه.

/// </summary>

Reboot,

/// <summary>

/// دریافت موقعیت فعلی دستگاه.

/// </summary>

GetLocation,

/// <summary>

/// تنظیم پیکربندی دستگاه.

/// </summary>

SetConfig,

/// <summary>

/// ارسال هشدار به دستگاه.

/// </summary>

SendAlert,

/// <summary>

/// شروع ردیابی.

/// </summary>

StartTracking,

/// <summary>

/// توقف ردیابی.

/// </summary>

StopTracking,

/// <summary>

/// بروزرسانی نرم‌افزار دستگاه.

/// </summary>

UpdateFirmware,

/// <summary>

/// قطع اتصال دستگاه.

/// </summary>

Disconnect,

/// <summary>

/// تست اتصال دستگاه.

/// </summary>

Ping

}

}

```

---

### \*\*2. توضیحات جامع\*\*

#### \*\*1. هدف:\*\*

- \*\*مدیریت انواع دستورات:\*\* این Enum برای دسته‌بندی انواع دستوراتی که به دستگاه‌ها ارسال می‌شوند استفاده می‌شود.

- \*\*استفاده در سراسر پروژه:\*\* این Enum در بخش‌های مختلفی از پروژه مانند کنترلرها، سرویس‌ها، مخازن، و مدل‌ها استفاده می‌شود.

#### \*\*2. مقادیر تعریف‌شده:\*\*

- \*\*Reboot:\*\* دستور راه‌اندازی مجدد دستگاه.

- \*\*GetLocation:\*\* دستور دریافت موقعیت فعلی دستگاه.

- \*\*SetConfig:\*\* دستور تنظیم پیکربندی دستگاه.

- \*\*SendAlert:\*\* دستور ارسال هشدار به دستگاه.

- \*\*StartTracking:\*\* دستور شروع ردیابی.

- \*\*StopTracking:\*\* دستور توقف ردیابی.

- \*\*UpdateFirmware:\*\* دستور بروزرسانی نرم‌افزار دستگاه.

- \*\*Disconnect:\*\* دستور قطع اتصال دستگاه.

- \*\*Ping:\*\* دستور تست اتصال دستگاه.

#### \*\*3. ویژگی‌ها:\*\*

- \*\*قابلیت توسعه:\*\* اگر نیاز به اضافه کردن نوع جدیدی از دستور وجود داشته باشد، به‌راحتی می‌توان آن را به این Enum اضافه کرد.

- \*\*مرکزیت:\*\* تمامی انواع دستورات در یک محل متمرکز قرار دارند و مدیریت آن‌ها آسان است.

---

### \*\*3. مثال استفاده\*\*

#### \*\*1. استفاده در مدل‌ها:\*\*

##### \*\*CommandDto.cs\*\*

```csharp

using SharedKernel.Enums;

namespace CommandService.Models

{

/// <summary>

/// مدل انتقال داده‌ها برای دستورات.

/// </summary>

public class CommandDto

{

/// <summary>

/// شناسه دستگاه.

/// </summary>

public string DeviceId { get; set; }

/// <summary>

/// نوع دستور.

/// </summary>

public CommandType CommandType { get; set; }

/// <summary>

/// وضعیت دستور.

/// </summary>

public CommandStatus Status { get; set; }

/// <summary>

/// زمان ایجاد دستور.

/// </summary>

public DateTime CreatedAt { get; set; }

/// <summary>

/// زمان اجرای دستور.

/// </summary>

public DateTime? ExecutedAt { get; set; }

}

}

```

#### \*\*2. استفاده در سرویس‌ها:\*\*

##### \*\*CommandService.cs\*\*

```csharp

using SharedKernel.Enums;

using CommandService.Models;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace CommandService.Services

{

/// <summary>

/// سرویس مدیریت دستورات.

/// </summary>

public class CommandService : ICommandService

{

private readonly ICommandRepository \_commandRepository;

public CommandService(ICommandRepository commandRepository)

{

\_commandRepository = commandRepository;

}

/// <summary>

/// ارسال دستور به دستگاه.

/// </summary>

public async Task SendCommandAsync(CommandDto commandDto)

{

if (commandDto.CommandType == CommandType.Reboot)

{

// ارسال دستور راه‌اندازی مجدد

Console.WriteLine($"Reboot command sent to device {commandDto.DeviceId}");

}

else if (commandDto.CommandType == CommandType.GetLocation)

{

// ارسال دستور دریافت موقعیت

Console.WriteLine($"GetLocation command sent to device {commandDto.DeviceId}");

}

await \_commandRepository.SaveCommandAsync(commandDto);

}

/// <summary>

/// دریافت دستورات مرتبط با یک دستگاه.

/// </summary>

public async Task<List<CommandDto>> GetCommandsByDeviceAsync(string deviceId, CommandType? commandType)

{

var commands = await \_commandRepository.GetCommandsByDeviceAsync(deviceId);

if (commandType.HasValue)

{

commands = commands.FindAll(c => c.CommandType == commandType.Value);

}

return commands;

}

}

}

```

#### \*\*3. استفاده در کنترلرها:\*\*

##### \*\*CommandsController.cs\*\*

```csharp

using Microsoft.AspNetCore.Mvc;

using SharedKernel.Enums;

using CommandService.Models;

using CommandService.Services;

using System.Threading.Tasks;

namespace CommandService.Controllers

{

/// <summary>

/// کنترلر مدیریت دستورات.

/// </summary>

[ApiController]

[Route("api/[controller]")]

public class CommandsController : ControllerBase

{

private readonly ICommandService \_commandService;

public CommandsController(ICommandService commandService)

{

\_commandService = commandService;

}

/// <summary>

/// ارسال دستور جدید.

/// </summary>

[HttpPost]

public async Task<IActionResult> SendCommand([FromBody] CommandDto commandDto)

{

await \_commandService.SendCommandAsync(commandDto);

return Ok("Command sent successfully.");

}

/// <summary>

/// دریافت دستورات مرتبط با یک دستگاه.

/// </summary>

[HttpGet("{deviceId}")]

public async Task<IActionResult> GetCommands(string deviceId, CommandType? commandType)

{

var commands = await \_commandService.GetCommandsByDeviceAsync(deviceId, commandType);

return Ok(commands);

}

}

}

\*\*4. مزایای استفاده از `CommandType`\*\*

1. \*\*مدیریت مرکزی:\*\* تمامی انواع دستورات در یک محل متمرکز قرار دارند.

2. \*\*جلوگیری از خطاهای تایپی:\*\* با استفاده از Enum، از خطاهای تایپی در کد جلوگیری می‌شود.

3. \*\*قابلیت توسعه:\*\* اضافه کردن یا تغییر انواع دستورات به‌راحتی امکان‌پذیر است.

4. \*\*استفاده مجدد:\*\* انواع دستورات در هر بخش از پروژه قابل استفاده هستند.

در ادامه، کد کامل و بهینه‌شده برای فایل `NotFoundException.cs` در پوشه `Exceptions` ارائه شده است. این فایل شامل یک استثنا (Exception) سفارشی است که زمانی رخ می‌دهد که یک منبع یا آیتم مورد نظر در سیستم پیدا نشود. این استثنا به‌صورت مرکزی مدیریت می‌شود تا از یکپارچگی و سازگاری در سراسر پروژه اطمینان حاصل شود.

1. NotFoundException.cs\*\*

using System;

using System.Runtime.Serialization;

namespace SharedKernel.Exceptions

{

/// <summary>

/// استثنا زمانی که یک منبع یا آیتم پیدا نشود.

/// </summary>

public class NotFoundException : Exception

{

/// <summary>

/// شناسه منبع مورد نظر.

/// </summary>

public string ResourceId { get; }

/// <summary>

/// نوع منبع مورد نظر.

/// </summary>

public string ResourceType { get; }

/// <summary>

/// سازنده پیش‌فرض.

/// </summary>

public NotFoundException()

{

}

/// <summary>

/// سازنده با پیام خطا.

/// </summary>

/// <param name="message">پیام خطا.</param>

public NotFoundException(string message)

: base(message)

{

}

/// <summary>

/// سازنده با مشخصات منبع.

/// </summary>

/// <param name="resourceType">نوع منبع.</param>

/// <param name="resourceId">شناسه منبع.</param>

public NotFoundException(string resourceType, string resourceId)

: this($"Resource of type '{resourceType}' with ID '{resourceId}' was not found.")

{

ResourceType = resourceType;

ResourceId = resourceId;

}

/// <summary>

/// سازنده با پیام خطا و خطای داخلی.

/// </summary>

/// <param name="message">پیام خطا.</param>

/// <param name="innerException">خطای داخلی.</param>

public NotFoundException(string message, Exception innerException)

: base(message, innerException)

{

}

/// <summary>

/// سازنده برای Serialization.

/// </summary>

/// <param name="info">اطلاعات Serialization.</param>

/// <param name="context">Context Serialization.</param>

protected NotFoundException(SerializationInfo info, StreamingContext context)

: base(info, context)

{

}

}

}

\*\*2. توضیحات جامع\*\*

\*\*1. هدف:\*\*

- \*\*مدیریت استثناها:\*\* این استثنا برای مدیریت خطاهایی است که زمانی رخ می‌دهند که یک منبع (مانند یک رکورد دیتابیس یا یک فایل) پیدا نمی‌شود.

- \*\*استفاده در سراسر پروژه:\*\* این استثنا در بخش‌های مختلفی از پروژه مانند کنترلرها، سرویس‌ها، مخازن، و مدل‌ها استفاده می‌شود.

2. ویژگی‌ها:\*\*

- \*\*ResourceId:\*\* شناسه منبع مورد نظر که پیدا نشده است.

- \*\*ResourceType:\*\* نوع منبع مورد نظر که پیدا نشده است.

- \*\*سازنده‌ها:\*\* شامل سازنده‌های مختلفی است که امکان ایجاد استثنای سفارشی را فراهم می‌کنند.

\*\*3. موارد پوشش‌داده‌شده:\*\*

- \*\*پیام خطا:\*\* پیام خطا به‌صورت پیش‌فرض یا سفارشی قابل تنظیم است.

- \*\*خطای داخلی:\*\* امکان افزودن خطای داخلی (Inner Exception) برای ردیابی علت اصلی خطا.

- \*\*Serialization:\*\* پشتیبانی از Serialization برای ذخیره‌سازی یا انتقال استثنا.

\*\*3. مثال استفاده\*\*

\*\*1. استفاده در مخازن:\*\*

\*\*RepositoryBase.cs\*\*

```csharp

using SharedKernel.Exceptions;

namespace Repository

{

public class RepositoryBase<T> where T : class

{

private readonly DbContext \_context;

public RepositoryBase(DbContext context)

{

\_context = context;

}

public T GetById(string id)

{

var entity = \_context.Set<T>().Find(id);

if (entity == null)

{

throw new NotFoundException(typeof(T).Name, id);

}

return entity;

}

}

}

\*\*2. استفاده در سرویس‌ها:\*\*

\*\*UserService.cs\*\*

using SharedKernel.Exceptions;

namespace UserService

{

public class UserService : IUserService

{

private readonly IUserRepository \_userRepository;

public UserService(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

public User GetUserById(string userId)

{

try

{

return \_userRepository.GetById(userId);

}

catch (NotFoundException ex)

{

// Log the exception or handle it as needed

Console.WriteLine($"Error: {ex.Message}");

throw;

}

}

}

}\*\*3. استفاده در کنترلرها:\*\*

\*\*UsersController.cs\*\*

using Microsoft.AspNetCore.Mvc;

using SharedKernel.Exceptions;

using UserService;

namespace WebApi.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class UsersController : ControllerBase

{

private readonly IUserService \_userService;

public UsersController(IUserService userService)

{

\_userService = userService;

}

[HttpGet("{id}")]

public IActionResult GetUser(string id)

{

try

{

var user = \_userService.GetUserById(id);

return Ok(user);

}

catch (NotFoundException ex)

{

return NotFound(new { Message = ex.Message });

}

}

}

}

\*\*4. مزایای استفاده از `NotFoundException`\*\*

1. \*\*مدیریت مرکزی:\*\* تمامی خطاهای مربوط به عدم پیدا شدن منابع در یک محل متمرکز قرار دارند.

2. \*\*جلوگیری از خطاهای تایپی:\*\* با استفاده از استثناهای سفارشی، از خطاهای تایپی در کد جلوگیری می‌شود.

3. \*\*قابلیت توسعه:\*\* اضافه کردن یا تغییر پیام‌ها و ویژگی‌های استثنا به‌راحتی امکان‌پذیر است.

4. \*\*استفاده مجدد:\*\* استثناهای سفارشی در هر بخش از پروژه قابل استفاده هستند.