**File Review**

**File:** Application/Application.csproj  
**Layer/Type:** Application – Project configuration  
**Status:** Reviewed  
**Tokens (approx.):** ~120

**🧾 ELI5**

This file is the **project configuration** for the Application layer. It defines dependencies, package references, and project references that Application needs to function.

**🎯 Purpose and Role**

* Establishes **dependencies** required for CQRS, DI, validation, and security.
* Connects Application layer to **Domain layer** via project reference.
* Configures **NuGet package dependencies** used across Application features.

**🔍 Detailed Breakdown**

* **Project reference:**
  + Domain/Domain.csproj → ensures Application depends on Domain, not the other way around (consistent with Clean Architecture).
* **Package references include:**
  + FluentValidation → for request validation.
  + MediatR and MediatR.Extensions.Microsoft.DependencyInjection → CQRS pattern implementation.
  + Microsoft.Extensions.Logging & Microsoft.Extensions.Options → logging and strongly-typed config.
  + BCrypt.Net-Next → password hashing.
  + System.IdentityModel.Tokens.Jwt → JWT handling for authentication.
* **Folders included in project build:**
  + Common/Configuration
  + DTOs/Dashboard, DTOs/Home, DTOs/Responses/Home
  + Mapping
  + Interfaces/Locking
  + Services/Etl

**⚠️ Error Handling & Validation**

* Not applicable directly to .csproj.
* Correctly scopes dependencies at Application layer.

**🔐 Security Review**

* ✅ Uses **BCrypt.Net-Next** for password hashing (stronger than SHA/MD5).
* ✅ Uses **System.IdentityModel.Tokens.Jwt** for secure token parsing.
* No insecure dependencies observed.

**⚡ Performance & Reliability**

* Dependencies are lightweight and standard for enterprise apps.
* MediatR and FluentValidation scale well when properly configured.
* Dependency injection ensures extensibility.

**📊 Observability**

* Logging support via Microsoft.Extensions.Logging.
* No specific observability library configured here (OpenTelemetry handled elsewhere per architecture).

**🧪 Testability & Coverage**

* MediatR + FluentValidation improve testability of Application layer.
* No direct test setup in this .csproj, but dependencies support unit/integration tests.

**🧹 Code Smells**

* **Info:** Some folder inclusions (DTOs/Dashboard, DTOs/Home) suggest possible coupling of UI-specific DTOs in Application layer. Should remain API-centric.

**🔧 Refactoring Suggestions**

* Ensure **DTOs are API-oriented** rather than UI-oriented to avoid mixing concerns.
  + Effort: **Medium**, Priority: **3**.
* Consider consolidating DTO folders if usage is fragmented.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Provides foundation for Application features.
* Breaking changes (e.g., removing MediatR) would ripple through handlers and validation.
* Dependencies must be kept up to date for security patches.

**✅ Confidence**

**High** – Full file available, dependencies align with Clean Architecture and ADRs.

**File Review**

**File:** Application/DependencyInjection.cs  
**Layer/Type:** Application – DI registration  
**Status:** Reviewed  
**Tokens (approx.):** ~180

**🧾 ELI5**

This file wires up the **Application layer** to the dependency injection (DI) container. It registers **MediatR handlers** and **FluentValidation validators** so that commands, queries, and validation rules work automatically.

**🎯 Purpose and Role**

* Provides a **centralised entry point** for configuring Application services.
* Ensures **CQRS pattern** via MediatR is properly registered.
* Ensures **validation rules** are automatically applied via FluentValidation.
* Keeps Application layer **independent of Web/Infrastructure** by exposing a single registration method.

**🔍 Detailed Breakdown**

* Declares a static class DependencyInjection with method:
* public static IServiceCollection AddApplicationServices(this IServiceCollection services)
* Registers:
  + **MediatR handlers** (services.AddMediatR(typeof(DependencyInjection).Assembly)).
  + **FluentValidation validators** (services.AddValidatorsFromAssembly(typeof(DependencyInjection).Assembly)).
* Returns the modified IServiceCollection for chaining.

**⚠️ Error Handling & Validation**

* No error handling (not required here).
* Assumes correct handler and validator discovery via reflection.

**🔐 Security Review**

* No direct secrets or sensitive values.
* Safe for DI bootstrap.

**⚡ Performance & Reliability**

* MediatR/FluentValidation assemblies are scanned **once at startup** – negligible overhead.
* Reflection-based scanning scales well for typical project sizes.

**📊 Observability**

* No logging here (correct for DI).
* Failures in DI registration will appear during startup, caught by host builder.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Call AddApplicationServices() → ensure MediatR handlers can be resolved.
2. **Positive:** Call AddApplicationServices() → ensure FluentValidation validators can be resolved.
3. **Edge:** Ensure method can be called multiple times without duplicate registrations.

**🧹 Code Smells**

* **Info:** Currently only registers MediatR and FluentValidation. As Application layer grows, risk of dumping unrelated services here.

**🔧 Refactoring Suggestions**

* Consider adding **regions or comments** for service categories (e.g., "CQRS", "Validation", "Pipelines").
  + Effort: **Quick Win**, Priority: **2**.
* Add **pipeline behaviours** registration here (e.g., logging, validation, performance tracking).
  + Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* Defines the **Application bootstrap contract**.
* Any change here affects Web project startup.
* Must remain stable for DI configuration consistency.

**✅ Confidence**

**High** – Full file reviewed, standard MediatR + FluentValidation setup, consistent with Clean Architecture and ADRs.

**File Review**

**File:** Application/Common/Result.cs  
**Layer/Type:** Application – Utility (Result wrapper)  
**Status:** Reviewed  
**Tokens (approx.):** ~280

**🧾 ELI5**

This file defines a generic **Result wrapper** used to indicate whether an operation was successful or failed. It contains either a value (on success) or an error message (on failure).

**🎯 Purpose and Role**

* Provides a **functional-style result type** (Result<T>).
* Reduces the need for exceptions in normal control flow.
* Standardises Application layer responses (especially in MediatR handlers).
* Encourages explicit error handling.

**🔍 Detailed Breakdown**

* **Properties:**
  + IsSuccess (bool) → true if operation succeeded.
  + Value (T?) → result payload (nullable).
  + Error (string?) → error message if failure.
* **Constructors:**
  + Private → ensures controlled creation through factory methods.
* **Factory methods:**
  + Success(T value) → returns successful result with value.
  + Failure(string error) → returns failed result with error.
  + Failure(string error, T? value) → failure with both error and optional payload.

**⚠️ Error Handling & Validation**

* Ensures controlled creation via static methods.
* Null safety:
  + Error is nullable → could be improved by enforcing non-empty strings.
  + Value can be null → intentional, but risk of misuse.

**🔐 Security Review**

* No secrets or sensitive values.
* Safe for usage across layers.

**⚡ Performance & Reliability**

* Very lightweight object.
* Value types boxed if used with generics, but impact negligible.
* Provides reliable pattern for success/failure handling.

**📊 Observability**

* Does not log errors; intended as carrier only.
* Logging handled by higher layers.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Result.Success("ok") → IsSuccess = true, Value = "ok".
2. **Negative:** Result.Failure("bad") → IsSuccess = false, Error = "bad".
3. **Edge:** Result.Failure("bad", null) → valid, but ensure null Value handled.
4. **Edge:** Result.Failure("") → check if empty error string allowed.

**🧹 Code Smells**

* **Low:** Value and Error nullable → risk of misuse.
* **Info:** Failure with both error and value may confuse semantics.

**🔧 Refactoring Suggestions**

* Enforce **non-empty Error strings**.
  + Effort: **Small**, Priority: **3**.
* Consider **separate types** for SuccessResult<T> and FailureResult<T>.
  + Effort: **Medium**, Priority: **4**.
* Add Match(successFunc, failureFunc) utility for functional consumption.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Used widely across Application and possibly Web layers.
* Changing its API would break MediatR handlers and validators.
* Must remain stable unless introducing non-breaking enhancements.

**✅ Confidence**

**High** – Classic result wrapper, fully available, aligned with best practices for Application layer.

**File Review**

**File:** Application/Common/TokenScopeExtensions.cs  
**Layer/Type:** Application – Utility (JWT parsing)  
**Status:** Reviewed  
**Tokens (approx.):** ~220

**🧾 ELI5**

This file adds a helper method to extract the **list of scopes** from a Xero OAuth2 access token (JWT). It parses the token, finds the scope claim, and returns a distinct list of values.

**🎯 Purpose and Role**

* Provides a safe, reusable way to extract **scopes** from Xero access tokens.
* Encapsulates parsing logic so Application and Web layers don’t duplicate it.
* Ensures consistent interpretation of Xero token claims.

**🔍 Detailed Breakdown**

* Defines static class TokenScopeExtensions.
* Method:
* public static List<string> GetScopes(this XeroOAuth2Token token)
* Logic:
  + Returns empty list if token or AccessToken is null/whitespace.
  + Uses JwtSecurityTokenHandler to parse the JWT.
  + Extracts scope or scp claims.
  + Splits values on space ' ' → supports multiple scopes.
  + Returns distinct list of scopes.

**⚠️ Error Handling & Validation**

* Gracefully handles null or invalid tokens.
* Does not throw if claim is missing → returns empty list instead.
* Potential risk:
  + Invalid JWT format could throw during ReadJwtToken (not caught here).

**🔐 Security Review**

* Token parsing done in-memory, no persistence.
* No exposure of sensitive values.
* Relies on token being validated before this method is called.

**⚡ Performance & Reliability**

* Token parsing is efficient for typical JWT size.
* LINQ operations are lightweight.
* No caching of parsed results → recomputed each call (acceptable unless very high throughput).

**📊 Observability**

* No logging here (correct).
* Errors in JWT parsing would bubble up → should be handled in calling code.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid JWT with "scope" claim → returns correct list.
2. **Positive:** Valid JWT with "scp" claim → returns correct list.
3. **Negative:** Invalid JWT string → expect exception.
4. **Edge:** Null token or null/empty AccessToken → returns empty list.
5. **Edge:** Multiple scopes in single claim → ensure split works.

**🧹 Code Smells**

* **Low:** Potential unhandled exception if AccessToken is invalid JWT.
* **Info:** Repeated parsing of same token could be costly if done often.

**🔧 Refactoring Suggestions**

* Wrap ReadJwtToken in try/catch to return empty list on invalid JWT.
  + Effort: **Small**, Priority: **4**.
* Consider caching scopes inside XeroOAuth2Token after first parse.
  + Effort: **Medium**, Priority: **3**.

**📜 Contracts & Compatibility**

* Directly tied to **Xero OAuth2 token contract**.
* Must remain compatible with Xero’s JWT structure (scope vs scp).

**✅ Confidence**

**High** – File fully available, clear purpose, aligned with Xero integration strategy.

**File Review**

**File:** Application/Common/Validation/ValidationExtensions.cs  
**Layer/Type:** Application – Validation Utilities  
**Status:** Reviewed  
**Tokens (approx.):** ~550

**🧾 ELI5**

This file defines **helper methods for FluentValidation** to simplify common validation tasks such as checking for empty GUIDs, validating emails, string lengths, and enforcing password rules.

**🎯 Purpose and Role**

* Provides **extension methods** for FluentValidation rule builders.
* Standardises validation rules across the Application layer.
* Encourages **consistent error messages** and enforcement of security-sensitive fields (e.g., passwords).

**🔍 Detailed Breakdown**

* Static class ValidationExtensions defines reusable validation rules:

**1. NotEmptyGuid<T>**

* Ensures a GUID field is not empty.
* Custom error message: "{fieldName} is required.".

**2. Email<T>**

* Ensures field is not empty.
* Validates format using .EmailAddress().
* Error messages: "Email is required." / "Enter a valid email address."

**3. NonEmptyStringWithMax<T>**

* Ensures non-empty string.
* Restricts maximum length.
* Error: "Field must be {maxLength} chars or fewer."

**4. NonEmptyStringWithLength<T>**

* Ensures string within min/max length bounds.
* Error: "Field must be between {minLength} and {maxLength} chars."

**5. Password<T>**

* Ensures non-empty string.
* Requires minimum length (default 8).
* Error: "Password must be at least {minLength} chars long."

**⚠️ Error Handling & Validation**

* FluentValidation handles error propagation.
* Provides consistent messages → avoids duplication.
* Possible improvement: add **null-safe handling** for string fields before length checks.

**🔐 Security Review**

* Strong emphasis on **password validation**.
* Current rules only enforce minimum length; do not check for complexity (uppercase, numbers, symbols).
* Depending on requirements, stronger password policies may be needed.

**⚡ Performance & Reliability**

* Rules are lightweight and efficient.
* Reusable across validators → avoids duplication and inconsistencies.

**📊 Observability**

* Error messages are explicit and user-friendly.
* No logging (correct for validation).

**🧪 Testability & Coverage**

Suggested test cases for each rule:

1. **NotEmptyGuid**: Guid.Empty → fail, valid Guid → pass.
2. **Email**: null, empty, invalid format, valid email.
3. **NonEmptyStringWithMax**: empty string, too long string, valid within limit.
4. **NonEmptyStringWithLength**: string shorter than min, longer than max, valid within range.
5. **Password**: empty string, too short string, meets min length.

**🧹 Code Smells**

* **Medium:** Password rule too lenient for production use.
* **Info:** Some duplication between string validation methods.

**🔧 Refactoring Suggestions**

* Add **configurable password policy** (min length + complexity options).
  + Effort: **Medium**, Priority: **5**.
* Consolidate NonEmptyStringWithMax and NonEmptyStringWithLength into single flexible method.
  + Effort: **Small**, Priority: **3**.
* Provide **unit tests** to lock down expected behaviour.
  + Effort: **Small**, Priority: **4**.

**📜 Contracts & Compatibility**

* Used by Application validators → centralised contract for validation.
* Changes here affect all validators → must avoid breaking changes.

**✅ Confidence**

**High** – File is fully available, aligns with FluentValidation best practices, consistent with security and domain needs.

**File Review**

**File:** Application/DTOs/Dashboard/DashboardViewModel.cs  
**Layer/Type:** Application – DTO (View Model)  
**Status:** Reviewed  
**Tokens (approx.):** ~220

**🧾 ELI5**

This file defines the **data structure for the dashboard screen**. It contains the user’s name, how many organisations they can access, recent API activity counts, and notifications to display.

**🎯 Purpose and Role**

* Serves as a **ViewModel DTO** for dashboard rendering.
* Provides aggregated, read-only data across multiple services.
* Composed by Application layer (handlers/services) before being passed to Web/UI.

**🔍 Detailed Breakdown**

* **Class:** DashboardViewModel (sealed).
* **Properties:**
  + CurrentUser (string, default "") → user display name.
  + TotalTenants (int) → number of organisations accessible.
  + RecentApiCalls (int) → number of recent API calls for user/orgs.
  + Notifications (List, default empty list) → list of alerts/notifications.
* **Constructor:**
  + Default constructor initialises class (no required args).

**⚠️ Error Handling & Validation**

* No validation (as expected for DTO).
* Defaults prevent null reference issues ("" and new List<string>()).

**🔐 Security Review**

* Contains only display data.
* No sensitive secrets.
* Must ensure **no PII leakage** (e.g., avoid including raw IDs or tokens).

**⚡ Performance & Reliability**

* Lightweight DTO.
* Reliably serialisable to JSON for APIs.
* Properties kept minimal → performant for dashboard usage.

**📊 Observability**

* No logging (correct for DTO).
* Observability must occur in service building this ViewModel.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Populate with typical user data → confirm correct JSON output.
2. **Negative:** Empty notifications list → still serialises correctly.
3. **Edge:** TotalTenants = 0 → verify dashboard displays gracefully.
4. **Edge:** RecentApiCalls large number → ensure UI handles scaling.

**🧹 Code Smells**

* **Info:** Mutable list property (Notifications) → could be modified after construction.

**🔧 Refactoring Suggestions**

* Use IReadOnlyList<string> for Notifications to enforce immutability.
  + Effort: **Small**, Priority: **3**.
* Consider constructor that requires CurrentUser, TotalTenants, RecentApiCalls.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Used by Web layer controllers to deliver dashboard JSON.
* Any changes to property names/types will affect frontend contract.

**✅ Confidence**

**High** – Fully available, straightforward DTO consistent with Clean Architecture.

**File Review**

**File:** Application/DTOs/Email/MailMessage.cs  
**Layer/Type:** Application – DTO (Email)  
**Status:** Reviewed  
**Tokens (approx.):** ~300

**🧾 ELI5**

This file defines the **structure of an email message** in RoadmApp. It contains recipients, subject, body (both HTML and plain text), and metadata like CC, BCC, and attachments.

**🎯 Purpose and Role**

* DTO used by Application layer to send emails via Infrastructure email providers.
* Provides a standardised, serialisable email object.
* Keeps email formatting and data consistent across the system.

**🔍 Detailed Breakdown**

* **Class:** MailMessage.
* **Properties:**
  + To (string) → recipient email (required).
  + Subject (string, default "") → email subject line.
  + HtmlBody (string, default "") → HTML version of message.
  + PlainTextBody (string, default "") → fallback for clients that don’t support HTML.
  + Cc (List, default empty).
  + Bcc (List, default empty).
  + Attachments (List, default empty).
* **Implementation Notes:**
  + Defaults prevent null issues.
  + Supports multi-recipient via CC/BCC.
  + Attachments list type suggests external Attachment class is defined elsewhere (likely Infrastructure).

**⚠️ Error Handling & Validation**

* No validation (as expected for DTO).
* Risks:
  + Empty To field would produce invalid email.
  + No enforcement of valid email addresses (relies on FluentValidation elsewhere).

**🔐 Security Review**

* Sensitive content (emails, attachments) → must avoid logging raw values.
* Attachments should be scanned/validated in Infrastructure layer.
* CC/BCC risk of **information leakage** if misused (e.g., exposing recipients).

**⚡ Performance & Reliability**

* Lightweight DTO.
* Attachments could affect performance if large (handled in Infrastructure).

**📊 Observability**

* No logging (correct for DTO).
* Failures to send logged in Infrastructure layer.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid To, Subject, HtmlBody → maps correctly to email provider.
2. **Negative:** Empty To → Infrastructure should reject.
3. **Edge:** Empty HtmlBody with non-empty PlainTextBody → still valid.
4. **Edge:** Multiple CC/BCC recipients.
5. **Edge:** With/without attachments.

**🧹 Code Smells**

* **Low:** To is single string → doesn’t support multiple primary recipients (should be List<string>).
* **Info:** No explicit validation for required fields.

**🔧 Refactoring Suggestions**

* Change To to List<string> to support multiple recipients.
  + Effort: **Medium**, Priority: **4**.
* Add domain-level validation (non-empty To, valid emails).
  + Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* Defines Application → Infrastructure contract for email sending.
* Changing To type from string to list would be a breaking change.
* Must remain compatible with Infrastructure email service provider.

**🗄️ Data Model Notes**

* Not persisted → transient DTO for message delivery.

**✅ Confidence**

**High** – Fully available, consistent with email DTO standards, requires stronger validation for production.

**File Review**

**File:** Application/DTOs/Organisation/OrganisationInfo.cs  
**Layer/Type:** Application – DTO (Organisation)  
**Status:** Reviewed  
**Tokens (approx.):** ~120

**🧾 ELI5**

This file defines a **data transfer object (DTO)** that carries display information about an organisation. It includes the organisation’s ID, name, number of users, and creation date.

**🎯 Purpose and Role**

* Provides a **lightweight representation** of an organisation for APIs or UI.
* Used in read operations (queries) where full domain entity is unnecessary.
* Allows safe transfer of organisation info without exposing sensitive fields.

**🔍 Detailed Breakdown**

* **Class:** OrganisationInfoDto (sealed).
* **Properties:**
  + OrganisationId (Guid).
  + OrganisationName (string, default "").
  + UserCount (int).
  + CreatedAt (DateTime).
* **Implementation Notes:**
  + Default string prevents null references.
  + Mutability via setters (standard for DTOs).

**⚠️ Error Handling & Validation**

* No validation (expected for DTO).
* Assumes higher layers enforce non-empty name and valid timestamps.

**🔐 Security Review**

* No sensitive data included.
* Safe for exposure in APIs.

**⚡ Performance & Reliability**

* Very lightweight.
* Efficient for listing organisations or displaying summary info.

**📊 Observability**

* No logging (correct for DTO).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Populate with valid values → confirm serialisation.
2. **Edge:** Empty OrganisationName → ensure UI handles gracefully.
3. **Edge:** UserCount = 0 → verify behaviour in API/UI.

**🧹 Code Smells**

* **Info:** Mutable properties may be changed after mapping.

**🔧 Refactoring Suggestions**

* Consider using record for immutability and value semantics.
  + Effort: **Small**, Priority: **2**.
* Optionally enforce non-empty OrganisationName via validation in Application layer.
  + Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* Defines API contract for organisation info.
* Changes to property names/types would break downstream consumers.

**✅ Confidence**

**High** – Fully available, typical DTO, consistent with project conventions.

**File Review**

**File:** Application/DTOs/Organisation/TenantDetailsDto.cs  
**Layer/Type:** Application – DTO (Organisation/Tenant)  
**Status:** Reviewed  
**Tokens (approx.):** ~140

**🧾 ELI5**

This file defines the **details of a tenant (organisation instance)**. It contains ID, name, connection status, scopes, and optional call statistics.

**🎯 Purpose and Role**

* Provides **organisation/tenant details** for UI or API responses.
* Used to display tenant connectivity, permissions (scopes), and activity stats.
* Enables Application layer to pass aggregated organisation data to Web layer.

**🔍 Detailed Breakdown**

* **Class:** TenantDetailsDto (sealed).
* **Properties:**
  + OrganisationId (Guid).
  + OrganisationName (string, default "").
  + IsConnected (bool).
  + Scopes (string, default "").
  + Stats (nullable CallStats) → reference to a response DTO (CallLogs).
* **Implementation Notes:**
  + Default string values prevent null issues.
  + Nullable Stats supports cases where logs aren’t available.

**⚠️ Error Handling & Validation**

* No validation here (standard for DTO).
* Risk: Scopes stored as a single string instead of structured list.

**🔐 Security Review**

* Exposes only non-sensitive organisation metadata.
* Scopes string should be sanitised before exposure to UI.

**⚡ Performance & Reliability**

* Lightweight DTO.
* Stats optional avoids unnecessary overhead.

**📊 Observability**

* No logging here.
* Observability handled in Application services producing this DTO.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Connected tenant with stats → populated DTO.
2. **Negative:** Not connected → IsConnected = false, Stats = null.
3. **Edge:** Empty Scopes string.

**🧹 Code Smells**

* **Medium:** Scopes as string instead of List<string>.
* **Info:** IsConnected + Stats dependency not explicit (could drift out of sync).

**🔧 Refactoring Suggestions**

* Change Scopes to List<string> for clarity and safer parsing.
  + Effort: **Medium**, Priority: **4**.
* Encapsulate IsConnected + Stats consistency via builder/service.
  + Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* Defines tenant detail contract for APIs.
* Changing Scopes type would be a **breaking change** to consumers.

**✅ Confidence**

**High** – File fully available, consistent with organisation/tenant DTO patterns.

**File Review**

**File:** Application/DTOs/Organisation/TenantListDto.cs  
**Layer/Type:** Application – DTO (Organisation/Tenant Collection)  
**Status:** Reviewed  
**Tokens (approx.):** ~90

**🧾 ELI5**

This file defines a **list of tenants (organisations)**. It wraps a collection of tenant details so APIs can return multiple organisations in a structured way.

**🎯 Purpose and Role**

* Provides a DTO for **returning multiple tenant records** at once.
* Used in queries where an organisation/user may have access to multiple tenants.
* Ensures a consistent JSON response structure instead of raw arrays.

**🔍 Detailed Breakdown**

* **Class:** TenantListDto (sealed).
* **Properties:**
  + Organisations (List, default new()).
* **Implementation Notes:**
  + Uses strong typing → avoids unstructured JSON arrays.
  + Default list prevents null reference errors.

**⚠️ Error Handling & Validation**

* No validation (expected for DTO).
* Relies on upstream service to populate properly.

**🔐 Security Review**

* Safe → contains only organisation metadata.
* No sensitive values included.

**⚡ Performance & Reliability**

* Lightweight wrapper.
* Efficient for serialisation/deserialisation.

**📊 Observability**

* No logging (correct for DTO).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Populate with multiple tenants → serialises correctly.
2. **Edge:** Empty list → ensure API returns [] instead of null.

**🧹 Code Smells**

* **Info:** Simple wrapper, but adds an extra nesting layer in API responses.

**🔧 Refactoring Suggestions**

* Evaluate whether wrapping is necessary vs returning plain list.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Defines API contract for multi-tenant responses.
* Removing wrapper class would be a **breaking change**.

**✅ Confidence**

**High** – File fully available, minimal wrapper DTO, aligns with API design patterns.

**File Review**

**File:** Application/DTOs/Organisation/UserOrganisationDto.cs  
**Layer/Type:** Application – DTO (Organisation/User Access)  
**Status:** Reviewed  
**Tokens (approx.):** ~160

**🧾 ELI5**

This file defines the **organisation details a user has access to**. It contains the organisation ID, name, and the OAuth scopes the user has within that organisation.

**🎯 Purpose and Role**

* Used to **list organisations a user belongs to** along with their granted scopes.
* DTO primarily for **UI population** (e.g., homepage tenant list).
* Ensures Application layer passes only relevant, non-sensitive organisation info.

**🔍 Detailed Breakdown**

* **Class:** UserOrganisationDto.
* **Properties:**
  + OrganisationId (Guid).
  + Name (string, default "").
  + Scopes (string, default "") → OAuth scopes granted for access.
* **Implementation Notes:**
  + Defaults prevent nulls.
  + Simple flattened structure for efficient transport.

**⚠️ Error Handling & Validation**

* No validation (expected for DTO).
* Scopes as string could lead to parsing inconsistencies.

**🔐 Security Review**

* Exposes only metadata → safe.
* Scopes string must be sanitised to avoid misinterpretation or injection.

**⚡ Performance & Reliability**

* Lightweight object.
* Efficient for API and JSON serialisation.

**📊 Observability**

* No logging here (correct).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Populate with valid organisation ID and name.
2. **Edge:** Empty Scopes → confirm UI handles gracefully.
3. **Edge:** User belongs to multiple organisations (collection of DTOs).

**🧹 Code Smells**

* **Medium:** Scopes as string (should be structured as list of strings).

**🔧 Refactoring Suggestions**

* Replace Scopes string with List<string>.
  + Effort: **Medium**, Priority: **4**.
* Optionally use record for immutability.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Defines user-organisation API contract.
* Changing Scopes type would be a **breaking change**.

**✅ Confidence**

**High** – File fully available, aligns with intended use for user-tenant mappings.

**File Review**

**File:** Application/DTOs/Requests/PasswordResetRequest.cs  
**Layer/Type:** Application – DTO (Request)  
**Status:** Reviewed  
**Tokens (approx.):** ~200

**🧾 ELI5**

This file defines the **request object for resetting a user’s password**. It includes the reset token ID, the token string itself, and the new password the user wants to set.

**🎯 Purpose and Role**

* Provides a standard request DTO for the **password reset workflow**.
* Ensures required fields (TokenId, Token, NewPassword) are present.
* Validated in Application layer before processing.

**🔍 Detailed Breakdown**

* **Class:** PasswordResetRequest.
* **Properties:**
  + TokenId (Guid) → unique identifier for the reset token.
  + Token (string, required, default "") → raw token string sent to user (email link).
  + NewPassword (string, required, min length 8, default "").
* **Data Annotations:**
  + [Required] → ensures values are provided.
  + [MinLength(8)] → enforces password length requirement.

**⚠️ Error Handling & Validation**

* Enforced via annotations (used by FluentValidation or ASP.NET model binding).
* Risks:
  + No complexity enforcement for NewPassword (only min length).
  + Token only checked for presence, not format.

**🔐 Security Review**

* **Critical:** Never log or persist NewPassword or raw Token.
* Password must be hashed in Infrastructure layer before storage.
* Token must be validated securely (timing-safe comparisons).

**⚡ Performance & Reliability**

* Lightweight request object.
* Reliability depends on Infrastructure handling secure hashing + token validation.

**📊 Observability**

* Should not log raw DTO values.
* Observability occurs at higher levels (success/failure of reset attempt).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid TokenId, Token, NewPassword length ≥ 8.
2. **Negative:** Empty Token → validation fails.
3. **Negative:** NewPassword < 8 characters → validation fails.
4. **Edge:** Very long NewPassword → ensure system caps correctly.

**🧹 Code Smells**

* **Medium:** Password validation too weak (length-only).

**🔧 Refactoring Suggestions**

* Add password complexity rules (uppercase, number, special char).
  + Effort: **Small**, Priority: **5**.
* Consider converting to record for immutability.
  + Effort: **Small**, Priority: **2**.

**📜 Contracts & Compatibility**

* Defines API request contract for password reset.
* Any change to validation rules must be coordinated with frontend.

**✅ Confidence**

**High** – File fully available, aligns with password reset flow, requires stronger password rules for production.

**File Review**

**File:** Application/DTOs/Requests/RunIngestRequest.cs  
**Layer/Type:** Application – DTO (Request)  
**Status:** Reviewed  
**Tokens (approx.):** ~80

**🧾 ELI5**

This file defines a **request object for triggering a data ingestion run** for a specific tenant. It only contains the tenant’s unique identifier.

**🎯 Purpose and Role**

* DTO for requesting a **manual ingestion run** for a given tenant.
* Used by Application services that orchestrate polling/ingestion.
* Keeps contract minimal and explicit.

**🔍 Detailed Breakdown**

* **Class:** RunIngestRequest.
* **Properties:**
  + TenantId (Guid) → identifies tenant for ingestion.
* **Implementation Notes:**
  + No defaults needed (Guid is value type).
  + Simple DTO wrapping a single value for clarity.

**⚠️ Error Handling & Validation**

* No validation here.
* Assumes TenantId must be a valid (non-empty) Guid, enforced elsewhere.

**🔐 Security Review**

* Safe – only carries tenant identifier.
* Must ensure caller is authorised to run ingestion for given tenant.

**⚡ Performance & Reliability**

* Extremely lightweight.
* No performance concerns.

**📊 Observability**

* No logging here (correct).
* Execution outcome observed in service layer.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid tenant GUID.
2. **Negative:** Guid.Empty → should be rejected at validation layer.

**🧹 Code Smells**

* **Info:** Single-property DTO could be replaced by inline parameter, but DTO keeps contract consistent.

**🔧 Refactoring Suggestions**

* Add [Required] annotation or FluentValidation rule for non-empty GUID.
  + Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* API contract for triggering ingestion.
* Changes would break API consumers.

**✅ Confidence**

**High** – Fully available, very simple request DTO, consistent with ingestion feature.

**File Review**

**File:** Application/DTOs/Requests/UserCreationRequest.cs  
**Layer/Type:** Application – DTO (Request)  
**Status:** Reviewed  
**Tokens (approx.):** ~160

**🧾 ELI5**

This file defines the **request object for creating a new user account**. It includes username, email, full name, and password.

**🎯 Purpose and Role**

* Standardises the input needed to **create a user account**.
* DTO used by Application layer handlers to validate and process user creation.
* Keeps frontend and backend aligned on required fields.

**🔍 Detailed Breakdown**

* **Class:** UserCreationRequest.
* **Properties:**
  + Username (string, default "").
  + Email (string, default "").
  + FullName (string, default "").
  + Password (string, default "").
* **Implementation Notes:**
  + No data annotations present here.
  + Likely validated via FluentValidation rules in Application/Common/Validation.

**⚠️ Error Handling & Validation**

* No built-in validation.
* Risks:
  + Empty username or email accepted if not validated elsewhere.
  + Password rule is unspecified (could be weak).

**🔐 Security Review**

* **Critical:** Password should never be logged.
* Must be hashed securely before persistence.
* Email and username should be validated to prevent duplicates and injection.

**⚡ Performance & Reliability**

* Lightweight DTO.
* Reliability depends on Application validation layer.

**📊 Observability**

* Should not be logged raw.
* Observability handled at higher layers (audit trail of account creation).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid Username, Email, Password → user created successfully.
2. **Negative:** Empty Username → validation fails.
3. **Negative:** Invalid email format → validation fails.
4. **Edge:** Very long FullName → enforce length limit.

**🧹 Code Smells**

* **Medium:** No validation annotations (risk of passing invalid data).
* **Low:** Default string.Empty masks missing fields.

**🔧 Refactoring Suggestions**

* Add [Required] and [EmailAddress] annotations, or enforce via FluentValidation.
  + Effort: **Small**, Priority: **4**.
* Add minimum password length requirement.
  + Effort: **Small**, Priority: **5**.

**📜 Contracts & Compatibility**

* Defines API request contract for user creation.
* Any changes must remain backward compatible with frontend.

**✅ Confidence**

**High** – Fully available, aligned with user account creation workflow, needs stronger validation/security.

**File Review**

**File:** Application/DTOs/Requests/UserRegistrationRequest.cs  
**Layer/Type:** Application – DTO (Request)  
**Status:** Reviewed  
**Tokens (approx.):** ~180

**🧾 ELI5**

This file defines the **request object for registering a new user account**. It contains username, email, password, and an optional full name.

**🎯 Purpose and Role**

* Provides a **standardised API contract** for user registration.
* Used when external users sign up (different from admin-driven creation).
* Ensures required fields are present and validated.

**🔍 Detailed Breakdown**

* **Class:** UserRegistrationRequest.
* **Properties:**
  + Username (string, [Required], [MinLength(3)]).
  + Email (string, [Required], [EmailAddress]).
  + Password (string, [Required], [MinLength(8)]).
  + FullName (string?, optional).
* **Implementation Notes:**
  + Data annotations provide validation metadata.
  + Enforces stronger rules than UserCreationRequest.

**⚠️ Error Handling & Validation**

* Enforced by [Required], [MinLength], [EmailAddress].
* Still lacks complexity validation for Password.

**🔐 Security Review**

* **Critical:** Password must never be logged or stored in plaintext.
* Must be hashed in Infrastructure before persistence.
* API should rate-limit to mitigate brute force on registration endpoint.

**⚡ Performance & Reliability**

* Lightweight request.
* Reliability depends on proper hashing and uniqueness checks for username/email.

**📊 Observability**

* No logging in DTO (correct).
* Registration attempts should be logged at service level (excluding sensitive values).

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid Username, Email, Password → accepted.
2. **Negative:** Username < 3 chars → validation fails.
3. **Negative:** Invalid email format → validation fails.
4. **Negative:** Password < 8 chars → validation fails.
5. **Edge:** Very long FullName.

**🧹 Code Smells**

* **Medium:** Still only enforces length on Password.
* **Info:** Duplication with UserCreationRequest (could be unified).

**🔧 Refactoring Suggestions**

* Add **password complexity rules**.
  + Effort: **Small**, Priority: **5**.
* Factor out shared properties into a **base user DTO** (reduce duplication with UserCreationRequest).
  + Effort: **Medium**, Priority: **3**.

**📜 Contracts & Compatibility**

* API contract for user self-registration.
* Stronger rules than UserCreationRequest may require consistent enforcement across endpoints.

**✅ Confidence**

**High** – File fully available, well-structured, requires stronger password rules and deduplication.

**File Review**

**File:** Application/DTOs/Requests/XeroEndpointRequest.cs  
**Layer/Type:** Application – DTO (Request, Xero Integration)  
**Status:** Reviewed  
**Tokens (approx.):** ~280

**🧾 ELI5**

This file defines the **request object for synchronising data from a single Xero API endpoint**. It specifies the endpoint, URL, required scopes, and pagination support.

**🎯 Purpose and Role**

* DTO for **configuring and executing Xero API syncs**.
* Provides flexibility in handling different endpoints (Invoices, Contacts, etc.).
* Used by Application layer services that orchestrate calls to Xero.

**🔍 Detailed Breakdown**

* **Class:** XeroEndpointRequest.
* **Properties:**
  + Name (string, default "") → endpoint identifier (e.g., "Invoices").
  + ApiUrl (string, default "") → relative API URL (e.g., /api.xro/2.0/Invoices).
  + Scopes (string?, optional) → required OAuth scopes for endpoint.
  + SupportsPagination (bool, default true) → indicates support for page query param.
  + SupportsOffset (bool, default false) → indicates support for offset param.
  + SupportsModifiedSince (bool, default true) → indicates support for If-Modified-Since header.
  + PageSize (int?, optional) → number of records per page (if applicable).
  + ResponseKey (string?, optional) → JSON key where records are found.
* **Implementation Notes:**
  + Defaults chosen based on typical Xero API behaviour.
  + Optional props allow endpoint-specific tuning.

**⚠️ Error Handling & Validation**

* No validation in DTO.
* Risks:
  + ApiUrl empty → invalid request.
  + PageSize not constrained → could exceed API limits.

**🔐 Security Review**

* DTO exposes only endpoint config.
* Must ensure **Scopes** string is validated to prevent misuse.
* API calls must still be authorised and signed securely.

**⚡ Performance & Reliability**

* DTO lightweight, but influences API performance via pagination.
* Incorrect values (e.g., missing pagination) could cause large data pulls.

**📊 Observability**

* DTO itself doesn’t log.
* Request/response logging must occur at service/infrastructure layer.

**🧪 Testability & Coverage**

Suggested test cases:

1. **Positive:** Valid endpoint (Invoices, 100 records/page).
2. **Negative:** Empty ApiUrl → validation fails.
3. **Edge:** PageSize = null → falls back to Xero defaults.
4. **Edge:** Toggle SupportsPagination/SupportsOffset.

**🧹 Code Smells**

* **Medium:** Scopes as string instead of list of scopes.
* **Low:** Multiple boolean flags → could be encapsulated in a capabilities object.

**🔧 Refactoring Suggestions**

* Convert Scopes to List<string>.
  + Effort: **Medium**, Priority: **4**.
* Replace booleans with an enum/flags struct to describe capabilities.
  + Effort: **Medium**, Priority: **3**.
* Add [Required] validation for ApiUrl.
  + Effort: **Small**, Priority: **4**.

**📜 Contracts & Compatibility**

* Defines API contract for ingestion jobs.
* Must remain compatible with Infrastructure sync services.

**✅ Confidence**

**High** – File fully available, well-structured, flexible for Xero integration.

**File Metadata**

* **File:** Application/DTOs/Responses/Auth/AuthenticatedUserResponse.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~120
* **Context:** Auth response DTO returned after successful login

**ELI5**

This file defines a response object that describes a successfully authenticated user — their ID, username, email, and which roles they have in each organisation.

**Purpose and Role**

* Serves as the **API response contract** for authentication endpoints.
* Consumed by **Web layer controllers** after login or identity refresh.
* Encapsulates **identity + authorisation** (roles per organisation).

**Detailed Breakdown**

public sealed class AuthenticatedUserResponse

{

public Guid UserId { get; set; }

public string Username { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public IReadOnlyDictionary<Guid, Guid> OrgRoles { get; set; } = new Dictionary<Guid, Guid>();

}

* **UserId** → unique identifier for the user (primary key from DB).
* **Username** → display name; default = string.Empty.
* **Email** → email address; default = string.Empty.
* **OrgRoles** → maps **OrganisationId → RoleId**, meaning per-tenant roles.
  + Uses IReadOnlyDictionary<Guid, Guid> for immutability.

**Error Handling & Validation**

* No explicit validation here (typical for DTO).
* Relies on upstream authentication logic to ensure UserId, Email, Username are valid.
* Defaults prevent null references.

**Security Review**

* ✅ No secrets or tokens are included.
* ⚠️ **PII risk**: Email is personally identifiable. Must not be logged at debug/error levels.
* ✅ Good practice: roles are expressed as IDs, not raw strings. (consistent with ADRs).

**Performance & Reliability**

* DTO is lightweight and efficient.
* Use of IReadOnlyDictionary avoids accidental mutation.

**Observability**

* No logging/metrics directly. Observability occurs at controller/middleware level.

**Testability & Coverage**

Recommended tests:

1. **Happy path:** Authenticated user with multiple organisations and roles.
2. **Edge case:** User with no organisations → empty OrgRoles.
3. **Edge case:** User with multiple orgs but same role ID across them.
4. **Negative:** Ensure Username/Email defaults to string.Empty when not set.

**Code Smells**

* **Info:** Email is plain string, no [EmailAddress] annotation (though validation probably occurs earlier).
* **Low:** OrgRoles being Guid → Guid loses semantic meaning (consumer must know which is OrgId vs RoleId).

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Replace Guid, Guid with a small typed DTO like OrganisationRoleDto { OrganisationId, RoleId } for readability.
* **Small (Priority 2):** Consider decorating Email with [EmailAddress] attribute if used in model validation.

**Contracts & Compatibility**

* This DTO is part of the **public API contract** for authentication responses.
* Changing OrgRoles type (to a DTO list) would be a **breaking change** → must version API if refactored.

**Data Model Notes**

* OrgRoles maps directly to join table UserOrgRole (Domain).
* Guid aligns with DB primary keys.

**Confidence**

**High** — full file content reviewed, DTO is straightforward.

**File Metadata**

* **File:** Application/DTOs/Responses/Auth/UserCreationResponse.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~150
* **Context:** Response returned after creating/registering a user

**ELI5**

This DTO describes the outcome of creating a new user account. It tells the client the new user’s ID, username, email, and whether further setup is required (email verification, password setup). It can also include an optional message.

**Purpose and Role**

* Used in **registration** or **admin-driven user creation** flows.
* Returned to clients to confirm what happened and what’s required next.
* Prevents exposing internal entities directly.

**Detailed Breakdown**

public sealed class UserCreationResponse

{

public Guid UserId { get; init; }

public string Username { get; init; } = string.Empty;

public string Email { get; init; } = string.Empty;

public bool RequiresEmailVerification { get; init; }

public bool RequiresPasswordSetup { get; init; }

public string? Message { get; init; }

}

* **UserId** → identity of new user.
* **Username** / **Email** → set from request. Defaults prevent nulls.
* **RequiresEmailVerification** → flags whether verification email must be clicked.
* **RequiresPasswordSetup** → flags whether user must set password before login.
* **Message** → optional friendly string, e.g., “Check your email to activate account.”

**Error Handling & Validation**

* No validation here (handled upstream by service/command handler).
* Defaults are safe.
* Nullable Message correctly marked optional.

**Security Review**

* ✅ No sensitive fields (passwords/tokens) included.
* ⚠️ Email is PII → same risk as before (must not be logged).
* ✅ No secrets exposed.

**Performance & Reliability**

* Very lightweight DTO.
* Immutable (init) setters → good for safety and thread-friendliness.

**Observability**

* No direct logging, handled at service/controller level.

**Testability & Coverage**

Recommended cases:

1. **Happy path:** User created requiring both email verification and password setup.
2. **Partial:** Only verification required, not password.
3. **Partial:** Only password setup required.
4. **Edge:** Neither required.
5. **Optional field:** Message null vs set.

**Code Smells**

* **Info:** Email field again lacks [EmailAddress] annotation.

**Refactoring Suggestions**

* **Quick Win (Priority 2):** Add [EmailAddress] attribute to align with validation best practices.
* **Small (Priority 3):** Consider converting Message into a **typed enum + message template** instead of arbitrary string (better for i18n and consistency).

**Contracts & Compatibility**

* Public API contract for user onboarding.
* Changing Message from string? to enum would be **breaking** unless versioned.

**Confidence**

**High** — complete file reviewed, clear intent.

**1. XeroAccessTokenDto.cs**

**File Metadata**

* **File:** Application/DTOs/Responses/Auth/XeroAccessTokenDto.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~170
* **Context:** Represents OAuth2 access/refresh token payload from Xero

**ELI5**

This DTO mirrors the JSON payload returned by Xero’s OAuth API when exchanging an authorization code or refreshing a token. It contains the access token, refresh token, expiry, and optional ID token.

**Purpose and Role**

* Used internally by the **Infrastructure/Auth service** to deserialize Xero’s token response.
* Should never be persisted or logged directly.
* Provides data to token management components (e.g., XeroTokenService).

**Detailed Breakdown**

public sealed record XeroAccessTokenDto(

[JsonPropertyName("access\_token")] string AccessToken,

[JsonPropertyName("refresh\_token")] string RefreshToken,

[JsonPropertyName("expires\_in")] int ExpiresIn,

[JsonPropertyName("id\_token")] string? IdToken

);

* **AccessToken** → short-lived bearer token.
* **RefreshToken** → long-lived token used to request new access tokens.
* **ExpiresIn** → seconds until expiry (typical 1800s = 30 min).
* **IdToken** (optional) → JWT with identity claims (only present if OpenID scope requested).

**Error Handling & Validation**

* Relies on correct JSON shape from Xero API.
* No local validation, but Infrastructure layer should validate null/empty.

**Security Review**

* ⚠️ **Critical**: Contains highly sensitive secrets (access + refresh tokens).
  + Must **never be logged**.
  + Must **only be stored encrypted** (ADR confirms use of Key Vault + encrypted persistence).
* ✅ Correctly marked as record — immutable once created.

**Performance & Reliability**

* Lightweight immutable DTO.
* Mapping is efficient with System.Text.Json.

**Testability & Coverage**

Suggested tests:

1. Deserialize valid payload with all fields.
2. Deserialize without id\_token.
3. Ensure invalid/malformed JSON fails gracefully.

**Code Smells**

* None in DTO itself.
* Security risk if mishandled downstream.

**Refactoring Suggestions**

* **Quick Win (Priority 5):** Add [JsonIgnore(Condition = WhenWritingDefault)] on IdToken to avoid serializing null.

**Contracts & Compatibility**

* Bound to **Xero API contract** → cannot change property names without breaking OAuth integration.

**Confidence**

**High** — file is clean, well-aligned with Xero’s OAuth spec.

**2. XeroAuthorizationSuccessResponse.cs**

**File Metadata**

* **File:** Application/DTOs/Responses/Auth/XeroAuthorizationSuccessResponse.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~90
* **Context:** Returned after completing Xero OAuth authorization flow

**ELI5**

This DTO is what the app sends back after Xero login succeeds. It tells the client when the token expires and which organisations (tenants) are linked.

**Purpose and Role**

* Serves as the **API response contract** for successful Xero login.
* Bridges the Infrastructure OAuth handshake with client applications.

**Detailed Breakdown**

public record XeroAuthorizationSuccessResponse(

DateTime TokenExpiryUtc,

List<XeroTenantInfo> Tenants

);

* **TokenExpiryUtc** → expiry timestamp of the Xero access token.
* **Tenants** → list of XeroTenantInfo domain entities (organisation context).

**Error Handling & Validation**

* No validation here.
* Depends on upstream services to ensure tenants are fetched.

**Security Review**

* ✅ Does not expose access/refresh tokens.
* ✅ Only includes expiry and tenant metadata (safe to return to client).
* ⚠️ Ensure no sensitive claims from IdToken are leaked here.

**Performance & Reliability**

* Lightweight response.
* Relies on DB/cache lookups of tenants being correct.

**Observability**

* Not applicable here; handled by higher layers.

**Testability & Coverage**

Cases:

1. One tenant returned.
2. Multiple tenants.
3. Edge: Empty tenant list.
4. Token expiry set close to now (test refresh triggers).

**Code Smells**

* **Info:** Uses domain entity XeroTenantInfo directly in DTO — mild layering violation (Application should ideally define its own DTO).

**Refactoring Suggestions**

* **Small (Priority 3):** Create a dedicated TenantInfoDto instead of reusing domain entity. This keeps layering clean and allows schema evolution.

**Contracts & Compatibility**

* Public API response → any changes must be versioned.
* Switching to a DTO for tenants would be a breaking change.

**Confidence**

**High** — file is clear, minimal risk.

**File Metadata**

* **File:** Application/DTOs/Responses/Config/EndpointInfo.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~100
* **Context:** DTO describing metadata for an API endpoint

**ELI5**

This object gives information about an API endpoint — its name, display name, API route, response key, and whether it supports pagination, offsets, or modified-since filtering.

**Purpose and Role**

* Exposes **configuration metadata** about available endpoints (likely Xero or internal APIs).
* Used by **UI rendering** (to show friendly names) and **service logic** (to know if pagination, offsets, or modified-since queries are supported).
* Helps consumers adapt dynamically to endpoint capabilities.

**Detailed Breakdown**

public sealed record EndpointInfo(

[JsonPropertyName("name")] string Name,

[JsonPropertyName("displayName")] string DisplayName,

[JsonPropertyName("api")] string Api,

[JsonPropertyName("responseKey")] string ResponseKey,

[JsonPropertyName("supportsPagination")] bool SupportsPagination,

[JsonPropertyName("pageSize")] int? PageSize,

[JsonPropertyName("supportsModifiedSince")] bool SupportsModifiedSince,

[JsonPropertyName("supportsOffset")] bool SupportsOffset

);

* **Name** → internal identifier.
* **DisplayName** → UI-friendly label.
* **Api** → the target API route.
* **ResponseKey** → key name in API payload (for deserialization).
* **SupportsPagination** / **PageSize** → indicates paging capability.
* **SupportsModifiedSince** → flag for incremental sync capability.
* **SupportsOffset** → flag for offset-based pagination.

**Error Handling & Validation**

* Relies on correct configuration source (JSON).
* Nullable PageSize → optional, must be checked before use.

**Security Review**

* ✅ No secrets, PII, or tokens.
* ✅ Pure metadata.

**Performance & Reliability**

* DTO is immutable (record).
* Serialization is efficient.
* If misconfigured, consumers could attempt unsupported queries → needs defensive checks in service logic.

**Observability**

* No direct logging; handled where endpoints are invoked.

**Testability & Coverage**

Cases to test:

1. Endpoint with pagination only.
2. Endpoint with modified-since only.
3. Endpoint supporting both.
4. Endpoint with PageSize = null.
5. Incorrect JSON (missing required fields).

**Code Smells**

* **Info:** All properties are raw strings/flags; no validation at DTO level.
* **Low:** Api could be confused with base URL vs path → naming could be clearer.

**Refactoring Suggestions**

* **Quick Win (Priority 2):** Rename Api → ApiRoute for clarity.
* **Small (Priority 3):** Replace booleans with a **capabilities enum/flags** to reduce proliferation of flags.

**Contracts & Compatibility**

* Public-facing DTO, likely part of API contract for endpoint discovery/config.
* Renaming or changing structure requires versioning.

**Confidence**

**High** — file is clear, consistent with config metadata patterns.

**File Metadata**

* **File:** Application/DTOs/Responses/Home/HomeDto.cs
* **Layer/Type:** Application / DTO (Response)
* **Status:** Reviewed
* **Tokens:** ~100
* **Context:** Data transfer object for rendering the user’s home page/dashboard

**ELI5**

This DTO provides the data needed to show a user’s home page: whether they’re connected to Xero, and which organisations they can access.

**Purpose and Role**

* Shapes the response for **home/dashboard API calls**.
* Lets UI know whether to show “Connect to Xero” prompts and which organisations to display.
* Central in onboarding and navigation UX.

**Detailed Breakdown**

public sealed class HomeDto

{

public bool IsConnected { get; set; }

public TenantListDto Organisations { get; set; } = new();

// Call statistics will be added later via TenantDetailsDto.Stats

}

* **IsConnected** → true if user is currently connected to Xero.
* **Organisations** → list of tenant/orgs accessible to the user (TenantListDto).
* Inline comment notes **call statistics** may be added in future.

**Error Handling & Validation**

* Defaults prevent null (Organisations = new TenantListDto()).
* No validation needed at this level.

**Security Review**

* ✅ No sensitive data exposed.
* ✅ Purely metadata about connectivity.
* ⚠️ Organisation details (names, IDs) are PII/business-sensitive — must not be logged.

**Performance & Reliability**

* Lightweight DTO.
* Scalability depends on how many organisations are returned (usually small set per user).

**Observability**

* No direct logging/metrics here.
* But this DTO is a candidate for UI telemetry (“connected vs not connected” rates).

**Testability & Coverage**

Recommended cases:

1. User connected to Xero with multiple organisations.
2. User connected with a single organisation.
3. User not connected (IsConnected = false, empty organisations).
4. Ensure Organisations defaults to non-null.

**Code Smells**

* **Info:** The TODO-style comment about call statistics suggests incomplete implementation.

**Refactoring Suggestions**

* **Small (Priority 3):** When adding statistics, prefer a nested DTO (e.g., OrganisationStatsDto) rather than mixing into TenantListDto.

**Contracts & Compatibility**

* Part of **public API contract** for home/dashboard endpoint.
* Adding new fields must preserve backward compatibility.

**Confidence**

**High** — file is simple and reviewed fully.

**1. EndpointIngestionResult.cs**

**ELI5**

Represents the result of ingesting data from a **single endpoint call** — how many rows inserted, whether it was up-to-date, and any error details.

**Breakdown**

public sealed record EndpointIngestionResult(

string EndpointName,

int RowsInserted,

bool WasUpToDate,

HttpStatusCode ResponseCode,

string? ErrorDetail

);

* Captures per-endpoint outcome.
* Provides clear summary for reporting/logging.

**Findings**

* ✅ Clean immutable record.
* ⚠️ ErrorDetail may expose raw API error — sanitize before logging.

**2. IngestionErrorSummary.cs**

**ELI5**

Summarizes a **single ingestion error** — endpoint, HTTP status, and error message.

**Breakdown**

public sealed class IngestionErrorSummary

{

public string EndpointName { get; init; }

public int HttpStatusCode { get; init; }

public string? ErrorDetail { get; init; }

}

**Findings**

* ✅ Lightweight, good for reporting failures.
* ⚠️ ErrorDetail again may leak raw provider errors (sanitize for UI).
* ✅ Uses immutable init-only properties.

**3. IngestionReport.cs**

**ELI5**

Provides a **summary report per endpoint/component** — total records processed, errors, timestamps, and skipped records.

**Breakdown**

public record IngestionReport(

string Component,

string Status,

int RecordsProcessed,

int ErrorsDetected,

DateTime Timestamp,

HttpStatusCode HttpStatusCode,

string? Message,

bool IsUpToDate = false,

int SkippedCount = 0

);

**Findings**

* ✅ Provides both **quantitative** (records, errors) and **qualitative** (status, message).
* ✅ Defaults for IsUpToDate and SkippedCount help avoid nulls.
* ⚠️ Status is raw string — consider enum to enforce valid values.

**4. IngestionResponse.cs**

**ELI5**

The **top-level ingestion response** — summarizes totals across all endpoints, includes per-endpoint reports, and error summaries.

**Breakdown**

public sealed class IngestionResponse

{

public Guid BatchId { get; init; }

public int TotalRecordsInserted { get; init; }

public int TotalSkipped { get; init; }

public IReadOnlyList<IngestionReport> Reports { get; init; }

public IReadOnlyList<IngestionErrorSummary> Errors { get; init; }

}

**Findings**

* ✅ Clean aggregation of results.
* ✅ Good separation: Reports (success/partial) vs Errors.
* ⚠️ Possible **large payload size** if many endpoints/errors — consider pagination or truncation in UI.

**Cross-Cutting Analysis**

**Error Handling & Validation**

* No nulls (defaults applied).
* Relies on ingestion services to populate values correctly.

**Security**

* ⚠️ ErrorDetail fields in multiple DTOs could contain sensitive raw API messages. Must be sanitized before exposure to clients.

**Performance**

* Lightweight objects, serialisation efficient.
* Risk of bloated IngestionResponse for large jobs — consider limiting detail.

**Testability**

Suggested test cases:

1. All endpoints succeed.
2. One endpoint fails (error summary present).
3. Mixed success/failure with partial inserts.
4. Empty ingestion (no endpoints).
5. Very large ingestion (stress test payload).

**Code Smells**

* **Low:** Status as free string → risks inconsistency.
* **Info:** Possible duplication between EndpointIngestionResult and IngestionReport — overlap in purpose.

**Refactoring Suggestions**

* **Small (Priority 3):** Replace Status string with enum IngestionStatus { Success, Failed, Partial, UpToDate }.
* **Medium (Priority 2):** Consolidate overlap between EndpointIngestionResult and IngestionReport to avoid confusion.
* **Quick Win (Priority 4):** Sanitize ErrorDetail before exposure in API response.

**Contracts & Compatibility**

* These DTOs are part of public API surface for ingestion results.
* Any renaming (e.g., Status → enum) is a breaking change → must version API.

**Confidence**

**High** — all four files fully reviewed, consistent patterns confirmed.

**1. ApiCallLogEntry.cs**

**ELI5**

Represents a **single API call log entry** — when it happened, which endpoint was called, how many rows were inserted, status code, success flag, and error message if failed.

**Breakdown**

public class ApiCallLogEntry

{

public DateTimeOffset? CallTime { get; set; }

public string? Endpoint { get; set; }

public int RowsInserted { get; set; }

public int? StatusCode { get; set; }

public bool Success { get; set; }

public string? ErrorMessage { get; set; }

}

**Findings**

* ✅ Clear, minimal log entry schema.
* ⚠️ ErrorMessage may leak raw provider errors → sanitize before exposing to UI.
* ✅ Nullable StatusCode handles cases where call didn’t reach API.

**2. CallStats.cs**

**ELI5**

Aggregates **statistics for API calls per organisation** — counts of successes, failures, rows inserted, and last call time.

**Breakdown**

public sealed class CallStats

{

public Guid OrganisationId { get; init; }

public string OrganisationName { get; set; } = string.Empty;

public DateTime LastCallUtc { get; init; }

public int SuccessCount { get; init; }

public int FailCount { get; init; }

public int RowsInserted { get; init; }

}

**Findings**

* ✅ Good for monitoring ingestion health.
* ✅ Immutable (init).
* ⚠️ OrganisationName is PII — must not be logged without masking.
* **Info:** Similar metrics also exist in IngestionReport → some overlap.

**3. TenantLogsDto.cs**

**ELI5**

Wraps logs for a **single tenant**, containing a collection of ApiCallLogEntryDto entries.

**Breakdown**

public sealed class TenantLogsDto

{

public List<ApiCallLogEntryDto> Logs { get; }

}

public sealed class ApiCallLogEntryDto

{

public DateTimeOffset? CallTime { get; set; }

public string? Endpoint { get; set; }

public int? StatusCode { get; set; }

public string? ErrorMessage { get; set; }

public bool Success { get; set; }

public int RowsInserted { get; set; }

}

**Findings**

* ✅ Provides tenant-scoped call history.
* ⚠️ Duplicates fields with ApiCallLogEntry. Suggest unifying to avoid duplication.

**4. XeroConnectionDto.cs**

**ELI5**

Represents a **single Xero tenant connection** — ID and human-friendly name.

**Breakdown**

public sealed record XeroConnectionDto(Guid TenantId, string TenantName);

**Findings**

* ✅ Minimal and immutable.
* ✅ Used for connection listing.
* ⚠️ Must be consistent with XeroTenantInfo domain entity.

**5. XeroRefreshResponseDto.cs**

**ELI5**

Represents the **response when refreshing Xero tokens** — access token, refresh token, expiry, and optional id\_token.

**Breakdown**

public sealed record XeroRefreshResponseDto(

[JsonPropertyName("access\_token")] string AccessToken,

[JsonPropertyName("refresh\_token")] string RefreshToken,

[JsonPropertyName("expires\_in")] int ExpiresIn,

[JsonPropertyName("id\_token")] string? IdToken

);

**Findings**

* ⚠️ **Critical security**: Contains OAuth secrets (access/refresh tokens). Must **never** be logged or returned to client in plaintext.
* ✅ Correct JSON mapping to Xero API contract.
* ⚠️ Risk of duplication with XeroAccessTokenDto — need clear separation (internal vs external use).

**Cross-Cutting Analysis**

**Error Handling**

* Defaults present, but no built-in validation.
* Error fields (ErrorMessage, ErrorDetail) risk leaking raw upstream messages.

**Security**

* ✅ Most DTOs safe (logs + stats).
* ⚠️ **High risk**: XeroRefreshResponseDto includes tokens. Must be internal-only and sanitized if ever exposed to API clients.

**Performance**

* Lightweight DTOs, efficient serialisation.
* Only risk: very large log collections in TenantLogsDto.

**Testability**

Cases to test:

1. Successful call with rows inserted.
2. Failed call with error.
3. Organisation with multiple call stats.
4. Empty logs list.
5. Refresh response missing id\_token.

**Code Smells**

* **Medium:** Duplication between ApiCallLogEntry and ApiCallLogEntryDto.
* **Critical:** Risk of exposing XeroRefreshResponseDto externally.

**Refactoring Suggestions**

* **Medium (Priority 5):** Unify ApiCallLogEntry and ApiCallLogEntryDto into one DTO.
* **Small (Priority 4):** Ensure XeroRefreshResponseDto is **Infrastructure-only**, not returned by Web API.
* **Quick Win (Priority 3):** Sanitize ErrorMessage before returning to clients.

**Contracts & Compatibility**

* Call logs are part of **public API contract** (used in monitoring/reporting).
* Any schema changes require versioning.
* Token DTOs must remain aligned with **Xero API contract**.

**Confidence**

**High** — all five files fully reviewed.

**File Metadata**

* **File:** Application/Features/Users/Queries/AuthenticateUserQuery.cs
* **Layer/Type:** Application / CQRS Query (MediatR)
* **Status:** Reviewed
* **Tokens:** ~90
* **Context:** Query object for authenticating a user

**ELI5**

This file defines a **query request** that carries the username/email and password so the system can check if a user is valid. Think of it as the envelope you hand to the authentication handler.

**Purpose and Role**

* Implements IRequest<Result<AuthenticatedUserResponse>> → meaning MediatR will handle it.
* Carries credentials (Username, Password) for downstream processing.
* Used by **query handler** (AuthenticateUserQueryHandler) to authenticate against persistence store.

**Detailed Breakdown**

public sealed class AuthenticateUserQuery : IRequest<Result<AuthenticatedUserResponse>>

{

public string Username { get; set; } = string.Empty;

public string Password { get; set; } = string.Empty;

}

* **Username** → input identifier (could be email or username).
* **Password** → plain text input; validation/verification must occur in handler.
* ✅ Defaults (string.Empty) avoid null issues.

**Error Handling & Validation**

* No validation logic here (query DTO).
* Handler must validate:
  + Non-empty username and password.
  + Ensure correct hashing/comparison with stored credentials.

**Security Review**

* ⚠️ **Critical:** Password field is plain text (expected at this layer). Must:
  + Never be logged.
  + Never be persisted.
  + Only used in **secure comparison** within Infrastructure/Domain layer.
* ✅ Query design aligns with ADR (CQRS via MediatR).

**Performance & Reliability**

* Very lightweight DTO.
* Bottlenecks will occur downstream in DB lookup + hash comparison, not here.

**Observability**

* No logging here (correct). Authentication handlers should log attempts (success/failure) with **redacted identifiers**.

**Testability & Coverage**

Recommended tests (in handler, not query itself):

1. Success → valid credentials return AuthenticatedUserResponse.
2. Failure → invalid username.
3. Failure → valid username but wrong password.
4. Edge → empty username or password (validation fail).
5. Security → ensure no password leaks in logs.

**Code Smells**

* **Critical:** Risk of password misuse if developer accidentally logs it.
* **Info:** No explicit validation on fields (assumed handled by FluentValidation or handler).

**Refactoring Suggestions**

* **Quick Win (Priority 5):** Ensure FluentValidation validator enforces [Required] + [MinLength] for both fields.
* **Small (Priority 4):** Consider renaming Username → UsernameOrEmail if both are accepted.

**Contracts & Compatibility**

* Public-facing query contract (part of auth flow).
* Changes to property names would break API → must be versioned.

**Confidence**

**High** — full file reviewed, straightforward CQRS query.

**File Metadata**

* **File:** Application/Features/Users/Queries/AuthenticateUserQueryHandler.cs
* **Layer/Type:** Application / CQRS Query Handler (MediatR)
* **Status:** Reviewed
* **Tokens:** ~200
* **Context:** Handles user authentication requests

**ELI5**

This file is the **engine** that checks if a user’s login credentials are valid. It asks the IUserService to authenticate the username/password, fetches the user’s roles, and then returns an AuthenticatedUserResponse if successful.

**Purpose and Role**

* Implements IRequestHandler<AuthenticateUserQuery, Result<AuthenticatedUserResponse>>.
* Acts as glue between the **MediatR query** and the **user service** that actually validates credentials.
* Maps domain results into an **Application DTO** (AuthenticatedUserResponse).

**Detailed Breakdown**

public sealed class AuthenticateUserQueryHandler

: IRequestHandler<AuthenticateUserQuery, Result<AuthenticatedUserResponse>>

{

private readonly IUserService \_userService;

public AuthenticateUserQueryHandler(IUserService userService) { ... }

public async Task<Result<AuthenticatedUserResponse>> Handle(

AuthenticateUserQuery request, CancellationToken cancellationToken)

{

var result = await \_userService.AuthenticateAsync(request.Username, request.Password, cancellationToken);

if (!result.IsSuccess || result.Value == null)

return Result<AuthenticatedUserResponse>.Failure(result.Error ?? "Authentication failed.");

var user = result.Value;

var orgRoles = await \_userService.GetUserRolesAsync(user.UserId, cancellationToken);

var output = new AuthenticatedUserResponse

{

UserId = user.UserId,

Username = user.Username,

Email = user.Email,

OrgRoles = orgRoles

};

return Result<AuthenticatedUserResponse>.Success(output);

}

}

**Error Handling & Validation**

* ✅ Uses Result<T> wrapper consistently → clear success/failure flow.
* ✅ Returns descriptive failure message when authentication fails.
* ⚠️ No explicit validation of Username/Password here — assumed upstream validation (FluentValidation).

**Security Review**

* ✅ Password never logged or persisted here (good).
* ✅ Delegates authentication to IUserService — ensures hashing/secure comparison happens in Infrastructure layer.
* ✅ User roles are fetched and returned as IDs (no raw names → reduces leakage).
* ⚠️ Ensure Error messages from \_userService are sanitized (no raw DB messages).

**Performance & Reliability**

* Two service calls (AuthenticateAsync, GetUserRolesAsync) — both async (good).
* ✅ CancellationToken propagated.
* ⚠️ Potential performance hit if roles query is not cached (should rely on caching in Infrastructure).

**Observability**

* No direct logging here (by design).
* Authentication attempts should be logged in IUserService with **correlation IDs**, not here.

**Testability & Coverage**

Recommended tests:

1. Success → valid credentials return correct AuthenticatedUserResponse.
2. Failure → invalid username.
3. Failure → valid username but wrong password.
4. Failure → service returns null user.
5. Edge → roles query returns empty dictionary.
6. Cancellation → operation cancelled before user fetch completes.

**Code Smells**

* **Low:** Error message fallback "Authentication failed." could be too generic for debugging but too specific could leak info. Balance needed.
* **Info:** Tight coupling to IUserService contract → but expected in CQRS.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Ensure IUserService.AuthenticateAsync enforces secure hashing/salting (Infrastructure concern, but critical).
* **Small (Priority 2):** Consider structured error codes instead of plain string messages.

**Contracts & Compatibility**

* Public API returns AuthenticatedUserResponse.
* Changes in shape of OrgRoles or AuthenticatedUserResponse → breaking API change.

**Confidence**

**High** — full file reviewed, aligns with MediatR + ADR guidance.

**File Metadata**

* **File:** Application/Features/Users/Commands/LoginUserCommandValidator.cs
* **Layer/Type:** Application / FluentValidation (Validator)
* **Status:** Reviewed
* **Tokens:** ~80
* **Context:** Validates login request fields

**ELI5**

This file makes sure that when a user tries to log in, they actually typed both a username/email and a password. If either is missing, it rejects the request before trying to authenticate.

**Purpose and Role**

* Uses **FluentValidation** to validate AuthenticateUserQuery.
* Prevents unnecessary calls to authentication service by checking inputs early.
* Ensures mandatory fields are present for login.

**Detailed Breakdown**

public sealed class LoginUserCommandValidator

: AbstractValidator<AuthenticateUserQuery>

{

public LoginUserCommandValidator()

{

RuleFor(x => x.Username)

.NotEmpty()

.WithMessage("Username or email is required.");

RuleFor(x => x.Password)

.NotEmpty()

.WithMessage("Password is required.");

}

}

* **RuleFor(x => x.Username)** → must not be empty.
* **RuleFor(x => x.Password)** → must not be empty.
* Messages provided for user feedback.

**Error Handling & Validation**

* ✅ Input validation is enforced.
* ✅ User-friendly error messages.
* ⚠️ No length/complexity validation here (may be enforced in registration, not login).

**Security Review**

* ✅ Prevents empty credential attempts (mitigates spam/DoS from empty requests).
* ⚠️ Still possible for brute force attacks with short passwords — rate limiting must be enforced at middleware/web layer.
* ✅ Password never logged here.

**Performance & Reliability**

* Very lightweight; runs before hitting the DB.
* ✅ Reduces load on authentication service.

**Observability**

* No direct logging (correct). Validation failures will be surfaced as validation exceptions handled centrally.

**Testability & Coverage**

Cases to test:

1. Both username and password provided → passes validation.
2. Empty username → fails with message.
3. Empty password → fails with message.
4. Both empty → fails with both messages.

**Code Smells**

* **Info:** This is named LoginUserCommandValidator, but it validates AuthenticateUserQuery. Slight naming inconsistency.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Rename file/class → AuthenticateUserQueryValidator for consistency.

**Contracts & Compatibility**

* Public API consumers will see validation messages.
* Changing messages or renaming fields would break UI/client assumptions.

**Confidence**

**High** — small, clear validator with no hidden issues.

**1. RegisterUserCommand.cs**

**ELI5**

This is the “form” a new user fills in when registering: username, email, password, and optionally their full name.

**Purpose and Role**

* Implements IRequest<UserCreationResponse>.
* DTO for carrying registration details to the handler.

**Breakdown**

public sealed class RegisterUserCommand : IRequest<UserCreationResponse>

{

public string Username { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string Password { get; set; } = string.Empty;

public string? FullName { get; set; }

}

* ✅ Defaults avoid nulls.
* ⚠️ Password is plain text (expected here, must be hashed downstream).

**Security**

* Critical: password must never be logged or persisted in plaintext.

**2. RegisterUserCommandHandler.cs**

**ELI5**

This is the “engine” that takes the registration form, asks the user service to create the user, and returns either success info or failure info.

**Purpose and Role**

* Implements IRequestHandler<RegisterUserCommand, UserCreationResponse>.
* Delegates persistence to IUserService.
* Uses ILogger for observability.

**Breakdown**

* Maps RegisterUserCommand → UserRegistrationRequest.
* Calls \_userService.RegisterAsync.
* On failure: logs warning and returns UserCreationResponse with error message, RequiresEmailVerification=false.
* On success: returns response with userId, username, email, and email verification requirement flag.

**Error Handling**

* ✅ Uses logging for failures.
* ✅ Returns structured UserCreationResponse.
* ⚠️ Currently swallows detailed error (returns generic "Registration failed." if result.Error is null).

**Security**

* ✅ Password passed securely to service (expected).
* ⚠️ Must ensure hashing + secure password policies enforced in Infrastructure.
* ⚠️ Logs must never include password or sensitive PII.

**Observability**

* ✅ Logs warnings on failure (good).
* ⚠️ Could benefit from structured logging with correlation ID.

**3. RegisterUserCommandValidator.cs**

**ELI5**

This file checks that a registration form is valid: username has length, email looks like an email, password meets requirements, and full name isn’t too long.

**Purpose and Role**

* Uses FluentValidation to validate RegisterUserCommand.

**Breakdown**

RuleFor(x => x.Username)

.NonEmptyStringWithLength(3, 100, "Username");

RuleFor(x => x.Email)

.Email();

RuleFor(x => x.Password)

.Password();

RuleFor(x => x.FullName)

.MaximumLength(200)

.WithMessage("Full name cannot be more than 200 characters.");

* ✅ Validates all key fields.
* Uses custom extensions: NonEmptyStringWithLength, Password() → likely defined in ValidationExtensions.

**Security**

* ✅ Password complexity enforced.
* ✅ Email validated.
* ✅ Prevents excessively long input (helps mitigate injection attempts).

**Cross-Cutting Analysis**

**Error Handling & Validation**

* Queries and commands enforce validation early.
* Handler gracefully handles service failures.

**Security**

* ⚠️ Password handling remains the most critical risk:
  + Ensure **never logged**.
  + Ensure **hashed + salted** before persistence.
* ✅ Validator enforces complexity.

**Performance**

* Lightweight validation.
* Handler delegates heavy work to services.

**Testability**

Cases to cover:

1. Valid registration succeeds → user created.
2. Duplicate username/email → failure response.
3. Invalid email format → validation fail.
4. Weak password → validation fail.
5. Full name > 200 chars → validation fail.

**Code Smells**

* **Info:** Generic error message on failure hides detail (tradeoff: less user guidance).
* **Low:** Handler mixes mapping + logging + service call; could delegate mapping.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Use structured error codes in response, not just messages.
* **Small (Priority 2):** Extract mapping from command → request into a mapper utility.
* **Medium (Priority 4):** Ensure correlation ID is included in logs for traceability.

**Contracts & Compatibility**

* Public API contract (UserCreationResponse) must stay stable.
* Changes to validation rules affect UX (must be communicated to clients).

**Confidence**

**High** — all three files fully reviewed, consistent with ADRs and validation patterns.

**1. LogoutUserCommand.cs**

**ELI5**

This is the instruction to log a user out. It just carries the user’s unique ID.

**Purpose and Role**

* Implements IRequest<bool>.
* Acts as a MediatR command envelope for logout operations.

**Breakdown**

public sealed class LogoutUserCommand : IRequest<bool>

{

public Guid UserId { get; set; }

}

* Only property: **UserId**.
* Used by handler to revoke tokens, sessions, etc.

**Security**

* ✅ No sensitive data exposed.
* ⚠️ Must ensure UserId belongs to current session (no impersonation).

**2. LogoutUserCommandHandler.cs**

**ELI5**

This is the “engine” that runs when a logout is requested. It logs that the user logged out and then returns true.

**Purpose and Role**

* Implements IRequestHandler<LogoutUserCommand, bool>.
* Handles cleanup and audit trail for logout.

**Breakdown**

public sealed class LogoutUserCommandHandler : IRequestHandler<LogoutUserCommand, bool>

{

private readonly ILogger<LogoutUserCommandHandler> \_logger;

public async Task<bool> Handle(LogoutUserCommand request, CancellationToken cancellationToken)

{

\_logger.LogInformation("User {UserId} has logged out.", request.UserId);

return await Task.FromResult(true);

}

}

* ✅ Uses logging for observability.
* ✅ Returns true (always).

**Cross-Cutting Analysis**

**Error Handling & Validation**

* No validation of UserId — assumed handled upstream (auth middleware).
* Always returns true, even if no actual cleanup occurred → may give false confidence.

**Security**

* ⚠️ Risk: If not paired with **token/session invalidation**, logout is just cosmetic logging.
* Must ensure tokens revoked in Infrastructure layer (e.g., Redis/Key Vault).
* ✅ No password/token exposure.

**Performance**

* Lightweight, synchronous.
* ✅ CancellationToken passed but not really needed (no I/O here).

**Observability**

* ✅ Logs logout with UserId.
* ⚠️ Better if it logged correlation ID + timestamp (though that may be middleware responsibility).

**Testability**

Cases:

1. Valid logout → returns true.
2. Ensure log entry is created.
3. Edge: Invalid/empty UserId (should still log, but check expectations).

**Code Smells**

* **Medium:** Handler does nothing but log and return true. Risk of giving illusion of proper logout handling when no token revocation occurs.

**Refactoring Suggestions**

* **Small (Priority 5):** Integrate token/session revocation into logout process.
* **Quick Win (Priority 3):** Add explicit check/logging if UserId is empty/invalid.

**Contracts & Compatibility**

* Public API returns bool.
* If switched to structured LogoutResponseDto, would be a breaking change.

**Confidence**

**High** — full files reviewed, straightforward but minimal implementation.

**1. ChangePasswordCommand.cs**

**ELI5**

This is the “form” a user fills in to change their password. It carries their user ID, the current password, and the new password they want to set.

**Purpose and Role**

* Implements IRequest<Result<bool>>.
* DTO for the password change operation.

**Breakdown**

public sealed class ChangePasswordCommand : IRequest<Result<bool>>

{

public Guid UserId { get; set; }

public string CurrentPassword { get; set; } = string.Empty;

public string NewPassword { get; set; } = string.Empty;

}

* ✅ Defaults prevent nulls.
* ⚠️ Both passwords are plain text here (expected at this layer).

**2. ChangePasswordCommandHandler.cs**

**ELI5**

This file checks if the user’s current password is correct, and if so, changes it to the new one. It logs the outcome.

**Purpose and Role**

* Implements IRequestHandler<ChangePasswordCommand, Result<bool>>.
* Uses IUserService to:
  + Authenticate with current password.
  + Change to new password.
* Logs both failure and success cases.

**Breakdown**

* Step 1: Verify current password with AuthenticateAsync.
* Step 2: If valid, call ChangePasswordAsync.
* Step 3: Log outcome and return result.

**Error Handling**

* ✅ Returns Result.Failure("Current password is incorrect.") when validation fails.
* ✅ Logs warning on incorrect current password.
* ✅ Logs error if change operation fails.

**3. ChangePasswordCommandValidator.cs**

**ELI5**

This file makes sure all password change requests are valid before they’re processed. It requires a user ID, current password, and new password, and ensures the new password is different from the old one.

**Purpose and Role**

* Uses FluentValidation to enforce password change rules.

**Breakdown**

RuleFor(x => x.UserId).NotEmptyGuid("User");

RuleFor(x => x.CurrentPassword)

.NotEmpty().WithMessage("Current password is required.");

RuleFor(x => x.NewPassword)

.Password("New password")

.NotEqual(x => x.CurrentPassword)

.WithMessage("New password must be different from the current password.");

* ✅ Enforces non-empty user ID and current password.
* ✅ Reuses custom .Password() rule (enforces complexity).
* ✅ Requires new password ≠ current password.

**Cross-Cutting Analysis**

**Security**

* ⚠️ Passwords are handled as plain strings (expected here). Must be:
  + Never logged.
  + Passed to hashing functions in Infrastructure layer.
* ✅ Validator enforces strong password policy.
* ✅ Handler validates current password before updating.

**Observability**

* ✅ Logs both success and failure with user ID.
* ⚠️ Logs must never contain actual passwords.

**Performance**

* Two service calls (AuthenticateAsync, ChangePasswordAsync).
* ✅ Both async and cancellation-aware.

**Testability**

Cases:

1. Valid → current password correct, new password accepted.
2. Failure → incorrect current password.
3. Failure → new password same as current password.
4. Failure → weak new password (validation).
5. Edge → empty user ID or missing fields.

**Code Smells**

* **Info:** Handler tightly couples authentication + update in one flow (but acceptable).
* **Low:** Generic error "Could not change password." may hide root cause.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Use structured error codes instead of raw strings.
* **Small (Priority 4):** Ensure correlation IDs included in logs for traceability.

**Contracts & Compatibility**

* Public API returns Result<bool>.
* Switching to richer response (with error codes) would be breaking.

**Confidence**

**High** — files reviewed fully, consistent with validation and CQRS design.

**1. ResetPasswordCommand.cs**

**ELI5**

This is the “form” that’s used when a user clicks a reset link. It contains the reset token ID, the token value, and the new password the user wants to set.

**Purpose and Role**

* Implements IRequest<Result<bool>>.
* DTO for password reset operation via reset token.

**Breakdown**

public sealed class ResetPasswordCommand : IRequest<Result<bool>>

{

public Guid TokenId { get; set; }

public string Token { get; set; } = string.Empty;

public string NewPassword { get; set; } = string.Empty;

}

* ✅ Contains both token ID and token string (two-factor verification).
* ⚠️ New password is plain text here (must be hashed later).

**2. ResetPasswordCommandHandler.cs**

**ELI5**

This is the “engine” that validates a password reset token, checks expiry, and if everything is valid, sets the user’s new password.

**Purpose and Role**

* Implements IRequestHandler<ResetPasswordCommand, Result<bool>>.
* Uses:
  + IPasswordResetTokenRepository to retrieve token record.
  + IUserService to update password.
* Logs results.

**Breakdown**

* Step 1: Fetch reset token from repository.
* Step 2: Validate:
  + Token exists.
  + Token not expired.
  + Token not already used.
  + Supplied token matches stored hash (BCrypt verify).
* Step 3: Call \_userService.ChangePasswordAsync with new password.
* Step 4: Mark token as used in repository.
* Step 5: Log success or failure.

**Error Handling**

* ✅ Covers null/missing token.
* ✅ Covers expired token.
* ✅ Covers reused token.
* ✅ Covers invalid token match.
* ✅ Returns Result.Failure with messages.

**3. ResetPasswordCommandValidator.cs**

**ELI5**

This makes sure the password reset form is valid before running. It requires the token ID, the token string, and a valid new password.

**Purpose and Role**

* Uses FluentValidation to enforce reset rules.

**Breakdown**

RuleFor(x => x.TokenId).NotEmptyGuid("Token ID");

RuleFor(x => x.Token)

.NotEmpty().WithMessage("Reset token is required.");

RuleFor(x => x.NewPassword)

.Password("New password");

* ✅ Ensures all fields present.
* ✅ Enforces password complexity.

**Cross-Cutting Analysis**

**Security**

* ✅ Validates token against hashed storage (BCrypt).
* ✅ Prevents reuse of token (marks as used).
* ✅ Prevents expired token use.
* ⚠️ New password must never be logged.
* ✅ Correct use of hashed verification aligns with ADR guidance.

**Observability**

* ✅ Logs warnings on failures and info on success.
* ⚠️ Log messages include token IDs and user IDs → PII risk if logs not secured.

**Performance**

* BCrypt verification is CPU-intensive but necessary for security.
* ✅ Async and cancellation-aware.

**Testability**

Cases:

1. Valid token → password reset succeeds.
2. Invalid token ID → failure.
3. Expired token → failure.
4. Token already used → failure.
5. Invalid token string → failure.
6. Weak new password → validation failure.

**Code Smells**

* **Low:** Handler is long, mixing validation + update. Could be split for clarity.
* **Info:** Uses plain string messages for errors.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Replace plain error strings with structured error codes.
* **Small (Priority 4):** Extract token validation logic into a helper/service to reduce handler length.
* **Medium (Priority 5):** Add correlation IDs to logs for traceability.

**Contracts & Compatibility**

* Public API returns Result<bool>.
* Any change to failure messaging affects client UX.

**Confidence**

**High** — full files reviewed, well-structured, aligns with secure password reset design.

**1. ForgotPasswordCommand.cs**

**ELI5**

This file represents the “forgot password” form. A user enters their email, and the system tries to send a reset link if the account exists.

**Purpose and Role**

* Implements IRequest<Result<bool>>.
* DTO for initiating password reset request.

**Breakdown**

public sealed class ForgotPasswordCommand : IRequest<Result<bool>>

{

public string Email { get; set; } = string.Empty;

}

* Only carries email address.
* Used by handler to trigger reset token creation.

**2. ForgotPasswordCommandHandler.cs**

**ELI5**

This is the “engine” that starts the password reset process. It checks whether an account exists for the given email, and if so, sends a reset link.

**Purpose and Role**

* Implements IRequestHandler<ForgotPasswordCommand, Result<bool>>.
* Delegates to IUserService.SendPasswordResetAsync(email, cancellationToken).

**Breakdown**

* Calls service to send reset link.
* Returns success if reset was triggered.
* Logs a warning on failure but still avoids leaking whether the email exists (security best practice).

**Error Handling**

* ✅ Uses Result<bool> wrapper for clear flow.
* ✅ Logs warnings with email (⚠️ PII risk if logs are not secured).
* ✅ Error message is generic: “If an account exists …”

**3. ForgotPasswordCommandValidator.cs**

**ELI5**

This file validates the “forgot password” form by ensuring the email is present and in a valid format.

**Purpose and Role**

* Uses FluentValidation.

**Breakdown**

RuleFor(x => x.Email).Email();

* ✅ Enforces non-empty, valid email.
* ✅ Prevents invalid requests from being processed.

**Cross-Cutting Analysis**

**Security**

* ✅ Does not reveal whether an email exists.
* ✅ Generic error messaging helps mitigate user enumeration.
* ⚠️ Logging the email in warnings could leak sensitive PII. Consider redacting or hashing for logs.

**Observability**

* ✅ Logs failures for monitoring.
* ⚠️ Should include correlation IDs for traceability.

**Performance**

* Lightweight validation.
* Main bottleneck is sending reset email via external provider.

**Testability**

Cases:

1. Valid email → success (reset link sent).
2. Unknown email → still returns success, logs warning.
3. Invalid email format → validation failure.
4. Empty email → validation failure.

**Code Smells**

* **Low:** Logs contain raw email address.
* **Info:** Relies entirely on service for sending email; no retry/backoff visible here.

**Refactoring Suggestions**

* **Quick Win (Priority 5):** Mask/redact email addresses in logs.
* **Small (Priority 3):** Add correlation ID to logs.
* **Medium (Priority 4):** Consider structured responses with error codes for observability.

**Contracts & Compatibility**

* Public API returns Result<bool>.
* Generic messaging is correct; must not change to “user not found” without careful consideration.

**Confidence**

**High** — flow is well-implemented and aligns with security practices for password reset initiation.

**1. SendPasswordResetCommand.cs**

**ELI5**

This is the “form” used to trigger sending a password reset email. It contains the user’s email and user ID.

**Purpose and Role**

* Implements IRequest<bool>.
* DTO for sending password reset email.

**Breakdown**

public class SendPasswordResetCommand : IRequest<bool>

{

public string Email { get; set; } = string.Empty;

public Guid UserId { get; set; }

}

* ✅ Captures both user ID and email (for extra verification).
* ⚠️ Email is PII — must not be logged.

**2. SendPasswordResetCommandHandler.cs**

**ELI5**

This is the “engine” that generates a reset token, stores it securely, and sends a reset link via email.

**Purpose and Role**

* Implements IRequestHandler<SendPasswordResetCommand, bool>.
* Dependencies:
  + IEmailService → to send reset email.
  + IPasswordResetTokenRepository → to persist reset tokens.
  + ILogger → for logging.

**Breakdown**

* Step 1: Generate cryptographically secure random token (Base64).
* Step 2: Hash token using BCrypt.
* Step 3: Create PasswordResetToken entity with expiry (1 hour).
* Step 4: Store hashed token in repository.
* Step 5: Construct callback URL containing raw token + token ID.
* Step 6: Send email with reset link.
* Step 7: Log success or failure.

**Error Handling**

* ✅ Logs and returns false if email sending fails.
* ✅ Wraps external service failures.
* ✅ Returns true only if everything succeeds.

**Cross-Cutting Analysis**

**Security**

* ✅ Uses **cryptographically secure random token** generation.
* ✅ Stores only **hashed token** (BCrypt), never raw.
* ✅ Sends raw token via email, but never stores it — correct practice.
* ⚠️ Logs include user ID and email address → must ensure logs are secure.
* ✅ Correctly prevents replay attacks (single-use tokens stored).

**Observability**

* ✅ Logs success and failure.
* ⚠️ Needs correlation IDs for audit trace.

**Performance**

* BCrypt hashing is slow but deliberate for security.
* ✅ Async repository and email sending calls.

**Testability**

Cases:

1. Valid user → reset email sent successfully.
2. Email service fails → returns false, logs error.
3. Invalid/missing email → failure.
4. Multiple requests for same user → new tokens generated each time.
5. Token stored properly (hashed).

**Code Smells**

* **Medium:** Handler is large, mixes token generation, storage, and email. Could be split into helpers.
* **Info:** Raw token returned only via email — correct, but ensure email template redacts logs.

**Refactoring Suggestions**

* **Quick Win (Priority 5):** Redact/mask email addresses in logs.
* **Small (Priority 4):** Extract token generation logic into a helper/utility class.
* **Medium (Priority 3):** Add structured error codes for better observability.

**Contracts & Compatibility**

* API returns bool.
* Changing return type to richer DTO (e.g., with error codes) would be breaking.

**Confidence**

**High** — robust implementation, follows best practices for secure reset tokens.

**1. CreateAdminUserCommand.cs**

**ELI5**

This is the “form” for creating an admin user. It includes email, and optionally username, full name, and password.

**Purpose**

* Implements IRequest<UserCreationResponse>.
* DTO for initiating admin creation.

**Breakdown**

* Email → required.
* Username, FullName, Password → optional.

**2. CreateAdminUserCommandHandler.cs**

**ELI5**

This is the “engine” that actually creates the admin. It passes the request to IUserService.InviteAdminAsync and returns a UserCreationResponse.

**Purpose**

* Implements IRequestHandler<CreateAdminUserCommand, UserCreationResponse>.
* Delegates persistence and business logic to IUserService.

**Breakdown**

* Maps CreateAdminUserCommand → UserCreationRequest.
* Calls \_userService.InviteAdminAsync.
* On failure: returns response with error message, requiring password setup and email verification.
* On success: returns UserCreationResponse with success message.

**Observations**

* ✅ Logs handled via ILogger.
* ⚠️ Error handling is limited — only string message returned.

**3. CreateAdminUserCommandValidator.cs**

**ELI5**

This file checks that the admin creation form is valid. Email is required, username must be ≥ 3 characters, and password (if supplied) must meet length rules.

**Purpose**

* Uses FluentValidation for admin creation validation.

**Breakdown**

RuleFor(x => x.Email).Email();

RuleFor(x => x.Username)

.MinimumLength(3).When(x => !string.IsNullOrWhiteSpace(x.Username))

.WithMessage("Username must be at least 3 characters.");

RuleFor(x => x.Password)

.MinimumLength(8).When(x => !string.IsNullOrWhiteSpace(x.Password))

.WithMessage("Password must be at least 8 characters.");

* ✅ Validates optional fields when provided.

**4. EditProfileCommand.cs**

**ELI5**

This is the “form” for editing a user’s profile (name, email).

**Purpose**

* Implements IRequest<Result<bool>>.
* DTO for profile edits.

**Breakdown**

* UserId → required.
* FullName and Email → new values.

**5. EditProfileCommandHandler.cs**

**ELI5**

This is the “engine” that updates a user’s profile. It calls IUserService.EditProfileAsync and logs the outcome.

**Purpose**

* Implements IRequestHandler<EditProfileCommand, Result<bool>>.
* Delegates to IUserService.

**Breakdown**

* Calls \_userService.EditProfileAsync(UserId, FullName, Email).
* Logs success or failure.
* Returns structured result.

**6. EditProfileCommandValidator.cs**

**ELI5**

This file validates the edit profile form. It requires a user ID, ensures full name is not too long, and validates email format.

**Purpose**

* Uses FluentValidation.

**Breakdown**

RuleFor(x => x.UserId).NotEmptyGuid("User ID");

RuleFor(x => x.FullName).NonEmptyStringWithMax(100, "Full name");

RuleFor(x => x.Email).Email();

* ✅ Prevents invalid inputs.

**Cross-Cutting Analysis**

**Security**

* ✅ No secrets exposed.
* ⚠️ Passwords (admin creation) must be hashed in Infrastructure layer.
* ⚠️ Emails are PII — must not be logged raw.

**Observability**

* ✅ Logs both success and failure.
* ⚠️ Should include correlation IDs for traceability.

**Performance**

* Lightweight. Main load is on DB service calls.

**Testability**

Cases:

1. Create admin with minimal info → success.
2. Create admin with short username → validation fail.
3. Create admin with weak password → validation fail.
4. Edit profile with invalid email → validation fail.
5. Edit profile with overly long name → validation fail.

**Code Smells**

* **Low:** Error messages are free text.
* **Info:** Handlers mix mapping, logging, and service call.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Replace free-text error strings with structured error codes.
* **Small (Priority 4):** Extract mapping into dedicated mappers for consistency.
* **Medium (Priority 2):** Add correlation IDs in logs.

**Contracts & Compatibility**

* CreateAdminUser returns UserCreationResponse (public API contract).
* EditProfile returns Result<bool> (less descriptive — may limit client feedback).

**Confidence**

**High** — all six files reviewed, consistent with CQRS patterns and FluentValidation best practices.

**1. AuthorizeXeroCommand.cs**

**ELI5**

This is the instruction that starts the Xero login process. It carries the authorisation code returned from Xero and the user ID.

**Purpose and Role**

* Implements IRequest<Result<AuthorizationResult>>.
* DTO for carrying the Xero OAuth2 callback code into the system.

**Breakdown**

public sealed class AuthorizeXeroCommand : IRequest<Result<AuthorizationResult>>

{

public string Code { get; }

public Guid UserId { get; }

public AuthorizeXeroCommand(string code, Guid userId)

{

Code = code;

UserId = userId;

}

}

* Code → authorisation code from Xero’s OAuth flow.
* UserId → RoadmApp user linking Xero tenant.

**2. AuthorizeXeroCommandHandler.cs**

**ELI5**

This is the “engine” that takes the Xero authorisation code, exchanges it for tokens, fetches the user’s tenants, and links them into the system.

**Purpose and Role**

* Implements IRequestHandler<AuthorizeXeroCommand, Result<AuthorizationResult>>.
* Orchestrates:
  + Token exchange with Xero.
  + Retrieving user’s tenants.
  + Linking tenants to RoadmApp user.
  + Returning result with success/failure and tenant info.

**Breakdown**

* Calls \_xeroTokenService.AuthorizeAsync(code, userId).
* Validates:
  + Tokens returned.
  + At least one tenant returned.
* Gets default role ID from \_accessService.
* Stores tokens via \_xeroTokenService.RetrieveTokenAsync.
* For each tenant:
  + Creates Unit of Work.
  + Calls ingestion service to onboard tenant data.
  + Ensures user is linked to tenant with default role.
* Returns AuthorizationResult containing successful and failed tenant IDs.

**Error Handling**

* ✅ Uses Result<T> wrapper.
* ✅ Logs failures per tenant.
* ✅ Returns structured result with lists of successes/failures.

**Cross-Cutting Analysis**

**Security**

* ✅ Secure token handling: tokens are only passed to \_xeroTokenService, which per ADR uses **Key Vault + Redis caching**.
* ✅ Correctly treats tokens as sensitive — not logged.
* ⚠️ Must ensure correlation IDs in logs when tenants fail to onboard.

**Observability**

* ✅ Logs failures with tenant ID and user ID.
* ⚠️ No telemetry on duration/performance of onboarding calls.

**Performance**

* Onboarding is potentially expensive (multiple tenant ingestion calls).
* ✅ Async calls with cancellation tokens.
* ⚠️ Sequential tenant processing may be slow with many tenants.

**Testability**

Cases:

1. Valid code → tokens + tenants returned → tenants onboarded.
2. Valid code but no tenants → failure result.
3. Tenant onboarding fails → appears in failed list.
4. Invalid code → failure.
5. User already linked to tenant → ensure idempotency.

**Code Smells**

* **Medium:** Handler is very long, mixing auth, onboarding, linking.
* **Info:** Error messages are free text.

**Refactoring Suggestions**

* **Small (Priority 3):** Extract tenant onboarding into a helper service.
* **Quick Win (Priority 4):** Add correlation IDs to logs.
* **Medium (Priority 2):** Consider batching tenant onboarding in parallel (careful with rate limits).

**Contracts & Compatibility**

* Returns AuthorizationResult (internal Application DTO).
* Any change in its structure requires updating client-side Xero flows.

**Confidence**

**High** — both files fully reviewed, implementation aligns with ADR for OAuth token handling and CQRS pattern.

**File Metadata**

* **File:** Application/Features/Xero/Commands/DisconnectXeroTenantCommand.cs
* **Layer/Type:** Application / CQRS Command (MediatR)
* **Status:** Reviewed
* **Tokens:** ~180
* **Context:** Handles disconnection of a Xero tenant from a RoadmApp user

**ELI5**

This file defines the command to disconnect a user from a Xero tenant and the handler that removes tokens and user access.

**Purpose and Role**

* Command carries **UserId** and **TenantId**.
* Handler:
  + Calls IXeroTokenService.DisconnectTenantAsync to remove Xero tokens.
  + Calls IUserService.RemoveUserOrgAccessAsync to remove local tenant access.
* Returns Result<Unit> indicating success or failure.

**Detailed Breakdown**

public record DisconnectXeroTenantCommand(Guid UserId, Guid TenantId)

: IRequest<Result<Unit>>;

public class DisconnectXeroTenantCommandHandler

: IRequestHandler<DisconnectXeroTenantCommand, Result<Unit>>

{

private readonly IXeroTokenService \_xeroTokenService;

private readonly IUserService \_userService;

public async Task<Result<Unit>> Handle(DisconnectXeroTenantCommand request, CancellationToken token)

{

var disconnectResult = await \_xeroTokenService.DisconnectTenantAsync(request.UserId, request.TenantId, token);

if (!disconnectResult.IsSuccess)

return Result<Unit>.Failure(disconnectResult.Error ?? "Disconnection failed.");

var removeAccessResult = await \_userService.RemoveUserOrgAccessAsync(request.UserId, request.TenantId, token);

if (!removeAccessResult.IsSuccess)

return Result<Unit>.Failure(removeAccessResult.Error ?? "Removal of access failed.");

return Result<Unit>.Success(Unit.Value);

}

}

**Error Handling & Validation**

* ✅ Uses Result<T> consistently.
* ✅ Distinguishes between token disconnection failure and user access removal failure.
* ⚠️ Fallback error messages are generic.

**Security Review**

* ✅ Correctly ensures both **token removal** and **user access revocation**.
* ⚠️ No logging here — must rely on IXeroTokenService/IUserService for audit logs.
* ✅ No secrets exposed directly.

**Performance & Reliability**

* Lightweight — two async service calls.
* ✅ CancellationToken propagated.
* ⚠️ No retry logic for transient errors (e.g., network blips to Xero API).

**Observability**

* ⚠️ No logging in handler itself → debugging requires downstream services to log.

**Testability & Coverage**

Cases:

1. Successful disconnect and access removal.
2. Token disconnection fails.
3. User access removal fails.
4. Edge: Invalid TenantId (no tenant found).
5. Edge: User has no access but disconnect requested.

**Code Smells**

* **Low:** Handler contains minimal logic; relies heavily on downstream services.
* **Info:** Lack of logging makes failures less transparent.

**Refactoring Suggestions**

* **Quick Win (Priority 3):** Add structured logging for audit trace (UserId, TenantId, correlation ID).
* **Small (Priority 4):** Enrich error messages with context (e.g., which step failed).
* **Medium (Priority 2):** Consider retry policy for external token service calls.

**Contracts & Compatibility**

* Returns Result<Unit>.
* Any switch to more detailed DTO (e.g., per-step outcome) would be a breaking change.

**Confidence**

**High** — complete file reviewed, consistent with CQRS and ADR patterns.

**1. XeroWebhookReceivedCommand.cs**

**ELI5**

This is the envelope for Xero webhook notifications. When Xero sends a payload to RoadmApp, this command carries the raw JSON string for processing.

**Purpose and Role**

* Implements IRequest<Unit>.
* Contains a single property: Payload (raw webhook JSON).

**Security**

* ⚠️ Webhook authenticity validation (signature check) must happen before this command is invoked.

**2. XeroWebhookReceivedCommandHandler.cs**

**ELI5**

This is the “engine” that processes Xero webhook payloads. It parses the JSON, validates it, and enqueues each event for processing.

**Purpose and Role**

* Implements IRequestHandler<XeroWebhookReceivedCommand, Unit>.
* Dependencies:
  + IXeroWebhookQueue → to enqueue events.
  + IXeroWebhookHistoryRepository → to persist raw payload for audit.
  + ILogger → for observability.

**Breakdown**

* Step 1: Save raw payload into history repository.
* Step 2: Parse JSON into JsonDocument.
* Step 3: Ensure events array exists.
* Step 4: For each event → enqueue to processing queue.
* Step 5: Log number of events processed.

**Error Handling**

* ✅ Catches JsonException → logs warning if payload invalid.
* ✅ Logs missing events array.
* ✅ Returns early if no valid events.

**Cross-Cutting Analysis**

**Security**

* ⚠️ The most important missing piece here: **webhook signature validation**.
  + Xero signs webhooks with HMAC; handler must ensure payload is authentic before processing.
* ✅ Raw payload persisted for audit (good for forensics).
* ⚠️ Ensure payloads don’t contain sensitive PII when stored long-term.

**Observability**

* ✅ Logs parsing failures, missing fields, and counts of events.
* ⚠️ Should include correlation ID for request tracing.

**Performance**

* Async enqueue per event.
* ⚠️ Events processed sequentially → may be slow with large payloads.

**Testability**

Cases:

1. Valid payload with multiple events → enqueued successfully.
2. Invalid JSON → logged warning, no processing.
3. Payload missing events → logged warning.
4. Empty events array → logged but still success.
5. Malicious payload (large JSON) → ensure DoS protection.

**Code Smells**

* **High:** No signature verification → potential spoofing attack vector.
* **Medium:** Handler mixes persistence, parsing, and queuing in one class.
* **Info:** Minimal schema validation of webhook event properties.

**Refactoring Suggestions**

* **Critical (Priority 5):** Implement Xero webhook HMAC signature validation before accepting payloads.
* **Small (Priority 3):** Extract parsing into a helper service.
* **Medium (Priority 2):** Add structured logging with correlation IDs.

**Contracts & Compatibility**

* Internal Application command, not directly API contract.
* Must remain consistent with Xero webhook schema.

**Confidence**

**High** — both files reviewed, design aligns with CQRS but missing **critical security validation**.

**File Review**

**File:** Application/Features/Organisation/Commands/AssignUsersCommand.cs  
**Layer/Type:** Application – MediatR Command DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~70

🧾 **ELI5**  
This file defines the request message for assigning a list of users to an organisation. It carries only the necessary IDs and relies on a handler to do the actual work.

🎯 **Purpose and Role**

* Represents a CQRS command (IRequest<Result<bool>>) sent through MediatR.
* Consumed by AssignUsersCommandHandler.
* Decouples intent (assign users) from implementation (organisation service).

🔍 **Detailed Breakdown**

* OrganisationId : Guid → identifies the target organisation.
* UserIds : List<Guid> = new() → list of users to assign, default empty to avoid null issues.
* Implements IRequest<Result<bool>> to produce a success/failure result.

⚠️ **Error Handling & Validation**

* No inline validation.
* Avoids null reference risk by initializing UserIds.
* Relies on FluentValidation (AssignUsersCommandValidator).

🔐 **Security Review**

* No sensitive data (only GUIDs).
* Safe for logging.

⚡ **Performance & Reliability**

* Lightweight DTO, serialisable.
* Safe for concurrent use.

📊 **Observability**

* No logging here (correct – belongs in handler).

🧪 **Testability & Coverage**  
Suggested test cases:

1. ✅ Valid OrgId + user list → accepted.
2. ❌ Empty OrgId → rejected by validator.
3. ❌ Null/empty user list → rejected by validator.

🧹 **Code Smells**

* None.

🔧 **Refactoring Suggestions**

* None required; clean DTO.

📜 **Contracts & Compatibility**

* Part of Application contract.
* Changing property names/types would break handler and external API contracts.

✅ **Confidence**  
High – simple DTO, full file visible.

**File Review**

**File:** Application/Features/Organisation/Commands/AssignUsersCommandHandler.cs  
**Layer/Type:** Application – MediatR Command Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~170

🧾 **ELI5**  
This file handles the “assign users” request by calling the organisation service and logging the outcome.

🎯 **Purpose and Role**

* Implements IRequestHandler<AssignUsersCommand, Result<bool>>.
* Delegates to IOrganisationService.AssignUsersAsync.
* Adds logging for both success and failure cases.

🔍 **Detailed Breakdown**  
Dependencies:

* IOrganisationService → business logic service for organisation updates.
* ILogger<AssignUsersCommandHandler> → structured logging.

Handle method:

* Calls AssignUsersAsync(OrganisationId, UserIds, cancellationToken).
* If failure: logs warning with OrgId + error, returns failure Result<bool>.
* If success: logs info with OrgId, returns success Result<bool>.

⚠️ **Error Handling & Validation**

* Uses result.IsSuccess check from service.
* Returns descriptive failure message.
* No retry logic – relies on service.

🔐 **Security Review**

* Logs OrgId and error messages only → safe.
* No secrets or PII exposed.

⚡ **Performance & Reliability**

* Async, non-blocking → good for scalability.
* Delegates concurrency and transactional integrity to service.

📊 **Observability**

* Logs success (LogInformation) and failure (LogWarning).
* Good traceability with OrgId context.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Success → service returns true, expect success result + info log.
2. Failure → service returns failure, expect failure result + warning log.
3. Edge → empty UserIds list still accepted? (covered by validator).
4. CancellationToken → ensure it stops execution.

🧹 **Code Smells**

* **Info:** String interpolation in logging ("Assign users failed for org {OrgId}: {Error}"). Could be structured log fields instead.

🔧 **Refactoring Suggestions**

* Use structured logging (LogWarning("Assign failed {OrgId} {Error}", orgId, error)) for better telemetry.
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Handler contract is stable under MediatR.
* Depends on service signature (AssignUsersAsync). Changes would cascade.

✅ **Confidence**  
High – full handler visible, aligns with ADRs and CQRS.

**File Review**

**File:** Application/Features/Organisation/Commands/AssignUsersCommandValidator.cs  
**Layer/Type:** Application – FluentValidation Validator  
**Status:** Reviewed  
**Tokens (approx.):** ~80

🧾 **ELI5**  
This file ensures that “assign users” requests always include a valid organisation ID and at least one user ID.

🎯 **Purpose and Role**

* Applies validation rules to AssignUsersCommand.
* Prevents malformed requests from reaching the handler.

🔍 **Detailed Breakdown**  
Rules:

* OrganisationId → must not be empty.
* UserIds → must not be null, must contain at least one item.

⚠️ **Error Handling & Validation**

* Rejects invalid commands early.
* Provides clear error messages:
  + "Organisation ID"
  + "User list is required."

🔐 **Security Review**

* No sensitive data.
* Error messages are safe to expose.

⚡ **Performance & Reliability**

* Lightweight.
* Validation is O(n) in UserIds length (minimal impact).

📊 **Observability**

* Validation failures surface through FluentValidation pipeline.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Empty OrgId → invalid.
2. Null UserIds → invalid.
3. Empty list → invalid.
4. Valid OrgId + multiple users → valid.

🧹 **Code Smells**

* None.

🔧 **Refactoring Suggestions**

* None required.

📜 **Contracts & Compatibility**

* Validation ensures stable input contract.
* Changes would directly affect API request acceptance.

✅ **Confidence**  
High – validator is small and clear.

**File Review**

**File:** Application/Features/Organisation/Commands/EditOrganisationCommand.cs  
**Layer/Type:** Application – MediatR Command DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~80

🧾 **ELI5**  
This file defines a request object that carries the details needed to edit an organisation, such as its ID and updated name.

🎯 **Purpose and Role**

* Represents an edit command in CQRS pattern (IRequest<Result<bool>>).
* Consumed by EditOrganisationCommandHandler.
* Decouples data transport (OrgId, new values) from execution logic.

🔍 **Detailed Breakdown**

* OrganisationId : Guid → identifies the organisation being edited.
* OrganisationName : string = string.Empty → new name (mutable property).
* Implements IRequest<Result<bool>>.

⚠️ **Error Handling & Validation**

* No inline validation.
* Default OrganisationName = string.Empty could lead to invalid updates if not validated elsewhere.
* Relies on EditOrganisationCommandValidator.

🔐 **Security Review**

* No secrets or PII.
* Safe for logging.

⚡ **Performance & Reliability**

* Lightweight DTO.
* Mutable properties allow runtime modification but unlikely problematic here.

📊 **Observability**

* None – correct for DTO.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid OrgId + non-empty name → valid.
2. Empty OrgId → rejected.
3. Empty/whitespace OrgName → rejected.
4. Long OrgName (edge of 100 chars) → valid.

🧹 **Code Smells**

* **Low:** Mutable property (OrganisationName) without validation defaults to empty string.

🔧 **Refactoring Suggestions**

* Consider making DTO properties init-only for immutability.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Contract between Application and API for organisation edits.
* Changing property names would ripple through handler and UI.

✅ **Confidence**  
High – file is short and clear.

**File Review**

**File:** Application/Features/Organisation/Commands/EditOrganisationCommandHandler.cs  
**Layer/Type:** Application – MediatR Command Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~170

🧾 **ELI5**  
This file performs the actual organisation update by calling the organisation service, then logs whether it succeeded or failed.

🎯 **Purpose and Role**

* Implements IRequestHandler<EditOrganisationCommand, Result<bool>>.
* Delegates work to IOrganisationService.EditOrganisationAsync.
* Logs results for auditing and debugging.

🔍 **Detailed Breakdown**  
Dependencies:

* IOrganisationService → service that applies edits.
* ILogger<EditOrganisationCommandHandler> → logs outcomes.

Handle method:

* Calls EditOrganisationAsync(request.OrganisationId, request.OrganisationName, cancellationToken).
* If failure: logs warning with OrgId + error, returns failure.
* If success: logs info with OrgId, returns success.

⚠️ **Error Handling & Validation**

* Handles IsSuccess from service correctly.
* Provides failure messages with error detail.
* No retry logic → appropriate to defer to service.

🔐 **Security Review**

* Logs OrgId and error reason only (safe).
* OrganisationName not logged → avoids accidental PII exposure.

⚡ **Performance & Reliability**

* Async and non-blocking.
* Service layer responsible for DB and concurrency safety.

📊 **Observability**

* Logs both failure (LogWarning) and success (LogInformation).
* OrgId included for correlation.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Success → service returns true, expect success result + info log.
2. Failure → service returns failure, expect failure result + warning log.
3. Empty OrgName (validator should reject).
4. CancellationToken cancels execution.

🧹 **Code Smells**

* **Info:** Logging uses string interpolation instead of structured parameters.

🔧 **Refactoring Suggestions**

* Use structured logging placeholders for better observability.
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Handler contract stable under MediatR.
* Depends on service signature → changes cascade.

✅ **Confidence**  
High – full file visible, consistent with ADRs.

**File Review**

**File:** Application/Features/Organisation/Commands/EditOrganisationCommandValidator.cs  
**Layer/Type:** Application – FluentValidation Validator  
**Status:** Reviewed  
**Tokens (approx.):** ~90

🧾 **ELI5**  
This file ensures that edit organisation requests include a valid organisation ID and a meaningful name.

🎯 **Purpose and Role**

* Validates input before handler execution.
* Prevents empty IDs and names.

🔍 **Detailed Breakdown**  
Rules:

* OrganisationId → must not be empty GUID.
* OrganisationName → must be non-empty and max length 100 chars.

⚠️ **Error Handling & Validation**

* Rejects invalid requests early.
* Custom error message: "Organisation name" for invalid names.

🔐 **Security Review**

* Safe – no secrets.
* Error messages are benign.

⚡ **Performance & Reliability**

* Lightweight validation.
* Enforces reasonable limits on OrgName.

📊 **Observability**

* Validation errors flow through FluentValidation.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Empty OrgId → invalid.
2. Null/empty OrgName → invalid.
3. OrgName length > 100 → invalid.
4. OrgName exactly 100 → valid.
5. Valid OrgId + OrgName → valid.

🧹 **Code Smells**

* None significant.

🔧 **Refactoring Suggestions**

* Consider stricter naming rules (e.g., disallow only whitespace).
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* Input contract validation for edit org API.
* Changes would affect API consumers.

✅ **Confidence**  
High – validator is short and clear.

**File Review**

**File:** Application/Features/Organisation/Queries/GetOrganisationInfoQuery.cs  
**Layer/Type:** Application – MediatR Query DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This file defines a query object used to request organisation details, either by specifying a particular organisation ID or by fetching the organisation associated with a given user.

🎯 **Purpose and Role**

* Implements IRequest<Result<OrganisationInfoDto>>.
* Defines the input contract for retrieving organisation info.
* Consumed by GetOrganisationInfoQueryHandler.

🔍 **Detailed Breakdown**

* UserId : Guid → identifies the requesting/associated user.
* OrganisationId : Guid → optional (default empty); if empty, handler may resolve organisation from user context.
* Constructor sets both values; OrganisationId defaults to Guid.Empty if not provided.

⚠️ **Error Handling & Validation**

* No inline validation.
* OrganisationId may be Guid.Empty, which requires handler to interpret as “resolve by user”.

🔐 **Security Review**

* Only carries IDs (safe).
* No secrets or sensitive data.

⚡ **Performance & Reliability**

* Lightweight DTO, no performance concerns.
* Relies on downstream service for data retrieval efficiency.

📊 **Observability**

* None (expected at DTO level).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid UserId + OrgId → direct lookup.
2. Valid UserId + empty OrgId → resolve via user association.
3. Empty UserId → should be rejected at validation/service level.

🧹 **Code Smells**

* **Low:** Overloaded constructor logic is implicit; clearer factory methods (ForUser, ForOrganisation) could improve readability.

🔧 **Refactoring Suggestions**

* Add named constructors for clarity:
  + GetOrganisationInfoQuery.ForOrganisation(Guid orgId)
  + GetOrganisationInfoQuery.ForUser(Guid userId)
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines part of Application query API.
* Changes here affect handler and API contract.

✅ **Confidence**  
High – DTO is simple and visible.

**File Review**

**File:** Application/Features/Organisation/Queries/GetOrganisationInfoQueryHandler.cs  
**Layer/Type:** Application – MediatR Query Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~160

🧾 **ELI5**  
This file handles requests to retrieve organisation information by calling the organisation service and returning the result.

🎯 **Purpose and Role**

* Implements IRequestHandler<GetOrganisationInfoQuery, Result<OrganisationInfoDto>>.
* Delegates to IOrganisationService.GetInfoAsync.
* Wraps service result in Result<OrganisationInfoDto>.

🔍 **Detailed Breakdown**  
Dependencies:

* IOrganisationService → provides organisation lookup.

Handle method:

* Calls GetInfoAsync(request.OrganisationId, cancellationToken).
* If service succeeds: returns Result.Success(result.Value).
* If service fails: returns Result.Failure(errorMessage).

⚠️ **Error Handling & Validation**

* Relies on service for validation.
* Gracefully returns failure with message if service fails.
* No exception handling beyond service result.

🔐 **Security Review**

* Safe: returns only OrganisationInfoDto.
* Does not log sensitive data (correct for query handler).

⚡ **Performance & Reliability**

* Async and non-blocking.
* Efficiency depends on service (likely backed by Dapper/PostgreSQL).
* Scales appropriately with MediatR pipeline.

📊 **Observability**

* No logging here.
* Traceability should be provided by service or higher-level middleware (consistent with ADRs).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Service returns success → expect Result.Success(dto).
2. Service returns failure → expect Result.Failure(error).
3. Invalid OrgId (Guid.Empty) → ensure resolved correctly.
4. CancellationToken cancels query.

🧹 **Code Smells**

* **Info:** Handler does not log failures (unlike command handlers). Consistency could be improved.

🔧 **Refactoring Suggestions**

* Consider adding logging of failures (at least warning level).
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Public Application query contract.
* Changes to return type or parameters would break consumer expectations.

✅ **Confidence**  
High – handler logic is visible and straightforward.

**File Review**

**File:** Application/Features/Polling/PollingFrequency.cs  
**Layer/Type:** Application – Enum (domain configuration helper)  
**Status:** Reviewed  
**Tokens (approx.):** ~40

🧾 **ELI5**  
This file defines an enum for supported polling frequencies (e.g., hourly, daily, weekly, or none). It’s used to configure how often data polling jobs should run.

🎯 **Purpose and Role**

* Provides a strongly-typed representation of polling intervals.
* Avoids using “magic strings” for scheduling.
* Consumed by polling commands, services, and schedulers.

🔍 **Detailed Breakdown**  
public enum PollingFrequency with values:

* Hourly
* Daily
* Weekly
* None

⚠️ **Error Handling & Validation**

* No validation here; relies on consumers to interpret correctly.
* None explicitly supported → useful for disabling polling, but needs careful handling downstream.

🔐 **Security Review**

* No secrets or sensitive data.
* Safe for logging and serialization.

⚡ **Performance & Reliability**

* Enums are lightweight and efficient.
* Provides compile-time safety against invalid values.
* Risk if downstream services don’t handle None properly (silent failure).

📊 **Observability**

* No logging here.
* Consumers may want to log when None disables polling.

🧪 **Testability & Coverage**  
Suggested test cases (at consumer level):

1. Ensure each enum maps correctly to scheduling logic (Hourly → 1h, Daily → 24h, Weekly → 7d).
2. Ensure None disables scheduling safely.
3. Edge: invalid enum value (from DB deserialization) → should fail gracefully.

🧹 **Code Smells**

* **Info:** Enum lacks explicit integer values. If persisted in DB, changing order could break compatibility.

🔧 **Refactoring Suggestions**

* Assign explicit integer values for durability (e.g., Hourly = 1, Daily = 2, etc.).
  + Effort: Quick Win, Priority: 4.
* Consider adding XML doc comments for each value to guide usage.
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* If persisted in database or sent via API, enum ordering matters.
* Safer to fix explicit values to prevent future reordering issues.

✅ **Confidence**  
High – full file reviewed, very small and clear.

**File Review**

**File:** Application/Features/Polling/Commands/SavePollingScheduleCommand.cs  
**Layer/Type:** Application – MediatR Command DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~200

🧾 **ELI5**  
This file defines a request object for saving or updating an organisation’s polling schedule (how often and when data is fetched).

🎯 **Purpose and Role**

* Represents a CQRS command (IRequest<Result<Unit>>).
* Consumed by SavePollingScheduleCommandHandler.
* Defines the polling configuration contract: organisation ID, frequency, optional runtime, and user ID.

🔍 **Detailed Breakdown**  
Properties:

* OrganisationId : Guid → required organisation reference.
* Frequency : PollingFrequency → enum (Hourly, Daily, Weekly, None).
* RunTime : TimeSpan? → optional daily run time.
* UserId : Guid → user saving the config.

Constructor: requires OrgId, Frequency, optional RunTime, and UserId.

⚠️ **Error Handling & Validation**

* No inline checks.
* Risk: RunTime could be invalid (e.g., combined with Hourly/Weekly).
* Validation deferred to UpdatePollingConfigCommandValidator.

🔐 **Security Review**

* Only IDs and scheduling info – safe for logging.
* No secrets or sensitive data.

⚡ **Performance & Reliability**

* Lightweight DTO.
* Complexity resides in downstream handler/service.

📊 **Observability**

* None – correct for DTO.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid OrgId + Hourly → valid.
2. Valid OrgId + Daily + RunTime → valid.
3. Empty OrgId → rejected by validator.
4. Invalid enum value → rejected.

🧹 **Code Smells**

* **Low:** Mutability not enforced; default constructor missing → misuse risk.

🔧 **Refactoring Suggestions**

* Make properties init-only for immutability.
  + Effort: Small, Priority: 3.
* Add guard clauses for RunTime (e.g., reject with Hourly).
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Defines input contract for polling config API.
* Changes here cascade to handler and persistence.

✅ **Confidence**  
High – fully visible, consistent.

**File Review**

**File:** Application/Features/Polling/Commands/SavePollingScheduleCommandHandler.cs  
**Layer/Type:** Application – MediatR Command Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~170

🧾 **ELI5**  
This file performs the actual save of a polling schedule by calling the polling settings service, handling success/failure, and returning a result.

🎯 **Purpose and Role**

* Implements IRequestHandler<SavePollingScheduleCommand, Result<Unit>>.
* Delegates persistence to IPollingSettingsService.UpsertAsync.
* Converts service response into MediatR Result<Unit>.

🔍 **Detailed Breakdown**  
Dependencies:

* IPollingSettingsService → handles persistence of polling config.

Handle method:

* Calls UpsertAsync(OrganisationId, Frequency, RunTime, cancellationToken).
* On success: returns Result.Success(Unit.Value).
* On failure: returns Result.Failure("Failed to save polling schedule: {Message}").
* Exceptions caught and returned as failures.

⚠️ **Error Handling & Validation**

* Uses try/catch → ensures unexpected exceptions return Result.Failure.
* Logs missing – handler does not record failures.

🔐 **Security Review**

* Safe – no sensitive data.
* Failure messages contain only generic error strings.

⚡ **Performance & Reliability**

* Async → scalable.
* Depends on service for DB performance and concurrency.
* No retry logic → left to service/resilience layer.

📊 **Observability**

* **Gap:** No logging (contrast with Organisation command handlers).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Service success → returns success.
2. Service failure → returns failure with error message.
3. Exception thrown → returns failure with exception message.
4. CancellationToken cancels execution.

🧹 **Code Smells**

* **Medium:** Missing structured logging of errors and correlation info.

🔧 **Refactoring Suggestions**

* Add logging (warning on failure, error on exception).
  + Effort: Quick Win, Priority: 3.
* Use structured logging placeholders.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Depends on IPollingSettingsService.UpsertAsync.
* Changes here ripple through service contract.

✅ **Confidence**  
High – handler fully visible.

**File Review**

**File:** Application/Features/Polling/Commands/UpdatePollingConfigCommandValidator.cs  
**Layer/Type:** Application – FluentValidation Validator  
**Status:** Reviewed  
**Tokens (approx.):** ~90

🧾 **ELI5**  
This file enforces validation rules for saving/updating polling configurations.

🎯 **Purpose and Role**

* Ensures commands have valid OrganisationId and Frequency.
* Prevents invalid schedules from reaching handler.

🔍 **Detailed Breakdown**  
Rules:

* OrganisationId → must not be empty.
* Frequency → must be a valid enum.

⚠️ **Error Handling & Validation**

* Rejects invalid OrgId.
* Rejects invalid frequency values.
* Does not validate RunTime against Frequency (potential gap).

🔐 **Security Review**

* No sensitive fields.
* Error messages safe to expose.

⚡ **Performance & Reliability**

* Lightweight validation.
* O(1) complexity.

📊 **Observability**

* Failures surfaced by FluentValidation pipeline.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Empty OrgId → invalid.
2. Invalid Frequency enum → invalid.
3. Valid OrgId + Hourly → valid.
4. Valid OrgId + Daily + RunTime → valid.
5. OrgId valid but RunTime nonsensical (e.g., with Hourly) → currently passes.

🧹 **Code Smells**

* **Medium:** Validation does not enforce relationship between Frequency and RunTime.

🔧 **Refactoring Suggestions**

* Add custom rule:
  + If Frequency == Hourly, RunTime must be null.
  + If Frequency == Daily, RunTime required.
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Defines API contract for polling config validity.
* Extending validation will affect client expectations.

✅ **Confidence**  
High – validator small and clear.

**File Review**

**File:** Application/Features/Ingestion/Queries/GetTenantLogsQuery.cs  
**Layer/Type:** Application – MediatR Query DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This file defines a query object used to fetch all ingestion/API call logs for a specific tenant (organisation) and user.

🎯 **Purpose and Role**

* CQRS query (IRequest<Result<TenantLogsDto>>).
* Consumed by GetTenantLogsQueryHandler.
* Encapsulates the request parameters: TenantId and UserId.

🔍 **Detailed Breakdown**

* TenantId : Guid → identifies organisation/tenant.
* UserId : Guid → identifies the requesting user (access control).
* Constructor assigns both values.

⚠️ **Error Handling & Validation**

* No inline validation.
* Assumes TenantId/UserId are valid.

🔐 **Security Review**

* Safe: only IDs.
* Sensitive only in access control context → handler/service must enforce.

⚡ **Performance & Reliability**

* Lightweight DTO.
* Performance depends on service/DB query downstream.

📊 **Observability**

* None here (correct).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid TenantId + UserId → expect logs.
2. Empty TenantId → rejected at service.
3. Unauthorized UserId → rejected at service.

🧹 **Code Smells**

* **Info:** No validation → relies fully on downstream.

🔧 **Refactoring Suggestions**

* Add FluentValidation (non-empty GUIDs).
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines query contract for retrieving tenant logs.
* Changes ripple to handler and API.

✅ **Confidence**  
High – simple DTO.

**File Review**

**File:** Application/Features/Ingestion/Queries/GetTenantLogsQueryHandler.cs  
**Layer/Type:** Application – MediatR Query Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This file retrieves tenant logs by calling a log service, maps them into DTOs, and returns them in a result wrapper.

🎯 **Purpose and Role**

* Implements IRequestHandler<GetTenantLogsQuery, Result<TenantLogsDto>>.
* Delegates work to ICallLogService.GetTenantLogsAsync.
* Maps raw logs into ApiCallLogEntryDto objects.

🔍 **Detailed Breakdown**  
Dependencies:

* ICallLogService → provides tenant logs.

Handle method:

* Calls GetTenantLogsAsync(TenantId, UserId, cancellationToken).
* Maps returned logs → ApiCallLogEntryDto with CallTime, Endpoint, StatusCode, ErrorMessage, Success, RowsInserted.
* Wraps mapped list in TenantLogsDto.
* Returns Result.Success(dto).

⚠️ **Error Handling & Validation**

* No explicit error handling → assumes service always returns logs or empty list.
* If service throws, exception bubbles (risk: no graceful failure).

🔐 **Security Review**

* Safe: returns logs, not secrets.
* Must ensure UserId is checked by service to prevent cross-tenant data leakage.

⚡ **Performance & Reliability**

* Mapping to DTO is O(n).
* DB query performance depends on indexes on TenantId, UserId, Timestamp.
* Missing error handling may cause reliability issues.

📊 **Observability**

* No logging of failures.
* Traceability depends on ICallLogService.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Service returns logs → expect mapped DTO.
2. Service returns empty list → expect empty DTO.
3. Service throws exception → handler should propagate (or ideally fail gracefully).
4. Unauthorized UserId → should reject at service.

🧹 **Code Smells**

* **Medium:** Missing try/catch → unlike other handlers (Org, Polling), no graceful failure path.
* **Low:** No logging on failure.

🔧 **Refactoring Suggestions**

* Add try/catch to wrap service errors into Result.Failure.
  + Effort: Small, Priority: 4.
* Add structured logging of failures (with TenantId).
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Public Application query contract.
* Changes here ripple to API consumers.

✅ **Confidence**  
High – handler logic is fully visible.

**File Review**

**File:** Application/Features/Ingestion/Dtos/IngestApiPayload.cs  
**Layer/Type:** Application – DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~110

🧾 **ELI5**  
This file defines the data payload used to return ingestion results, including total rows inserted, detailed endpoint results, and error summaries.

🎯 **Purpose and Role**

* Represents ingestion outcome details.
* Returned from ingestion-related operations to API consumers.
* Helps track both successful inserts and errors.

🔍 **Detailed Breakdown**

**IngestApiPayload**

* TotalInserted : int → total rows inserted across endpoints.
* Reports : IReadOnlyList<EndpointIngestionResult> → detailed results per endpoint.
* Errors : List<ErrorSummary> → collection of error summaries.

**ErrorSummary (nested class)**

* EndpointName : string? → name of the endpoint where error occurred.
* Code : int → error code.
* ErrorDetail : string? → human-readable error message.

⚠️ **Error Handling & Validation**

* No inline validation.
* Risk: ErrorDetail could contain sensitive details if not filtered upstream.
* Assumes TotalInserted aligns with sum of reports (not enforced).

🔐 **Security Review**

* Potential exposure of error details. Must ensure:
  + No sensitive data (tokens, connection strings) included in ErrorDetail.
  + Sanitization happens at ingestion service layer.

⚡ **Performance & Reliability**

* DTO is lightweight.
* Safe for serialization.
* Reports as IReadOnlyList ensures immutability for endpoint results.
* Errors is a mutable List<ErrorSummary> → risk of post-construction modification.

📊 **Observability**

* DTO is purely a transport object.
* Logging/telemetry happens upstream.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Payload with multiple endpoint results → correct TotalInserted.
2. Error summaries present → returned intact.
3. Empty Reports/Errors → still valid object.
4. Large number of Reports → ensure no serialization performance issues.

🧹 **Code Smells**

* **Low:** TotalInserted may drift out of sync with Reports count.
* **Info:** Mixed mutability: Reports is read-only, but Errors is mutable.

🔧 **Refactoring Suggestions**

* Consider making Errors also IReadOnlyList<ErrorSummary>.
  + Effort: Small, Priority: 3.
* Add constructor/factory enforcing consistency:
  + e.g., TotalInserted = Reports.Sum(r => r.RowsInserted).
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Defines part of public API surface for ingestion reporting.
* Changes to property names or types would break client integrations.

✅ **Confidence**  
High – full file visible, straightforward DTO.

**File Review**

**File:** Application/Features/Roles/Commands/AssignRoleCommand.cs  
**Layer/Type:** Application – MediatR Command DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~70

🧾 **ELI5**  
This file defines the request object for assigning a role to a user within an organisation.

🎯 **Purpose and Role**

* CQRS command (IRequest<Result<bool>>).
* Consumed by AssignRoleCommandHandler.
* Provides the minimal contract: UserId, OrganisationId, RoleId.

🔍 **Detailed Breakdown**  
Properties:

* UserId : Guid → target user.
* OrganisationId : Guid → tenant context.
* RoleId : Guid → role to assign.

⚠️ **Error Handling & Validation**

* No inline validation.
* Relies on FluentValidation.

🔐 **Security Review**

* Only IDs (safe).
* Security depends on handler/service verifying authorisation.

⚡ **Performance & Reliability**

* Lightweight DTO, no issues.

📊 **Observability**

* None (correct).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid UserId + OrgId + RoleId → accepted.
2. Empty UserId → rejected.
3. Empty OrgId → rejected.
4. Empty RoleId → rejected.

🧹 **Code Smells**

* None – clean DTO.

🔧 **Refactoring Suggestions**

* Could be immutable (init props).
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines contract for role assignment API.
* Changes ripple to handler and API consumers.

✅ **Confidence**  
High – simple DTO.

**File Review**

**File:** Application/Features/Roles/Commands/AssignRoleCommandHandler.cs  
**Layer/Type:** Application – MediatR Command Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This file assigns a role to a user by calling the access service. It logs success and failure, returning a result accordingly.

🎯 **Purpose and Role**

* Implements IRequestHandler<AssignRoleCommand, Result<bool>>.
* Delegates to IAccessService.AssignRoleAsync.
* Adds structured logging of success and failure.

🔍 **Detailed Breakdown**  
Dependencies:

* IAccessService → handles user-role assignment.
* ILogger<AssignRoleCommandHandler> → logs outcomes.

Handle method:

* Calls AssignRoleAsync(UserId, OrganisationId, RoleId, cancellationToken).
* On failure: logs warning and returns Result.Failure("Role assignment failed.").
* On success: logs info and returns Result.Success(true).

⚠️ **Error Handling & Validation**

* Checks result.IsSuccess.
* Logs failure and returns descriptive message.
* No exception handling – assumes service manages errors.

🔐 **Security Review**

* Logs only IDs, not sensitive data.
* Authorisation enforcement must be inside service.

⚡ **Performance & Reliability**

* Async, non-blocking.
* Reliability depends on IAccessService.

📊 **Observability**

* Good: logs both success (Info) and failure (Warning).
* Logs include UserId and OrgId for traceability.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Service success → success result + info log.
2. Service failure → failure result + warning log.
3. Invalid RoleId → should fail at validation/service.
4. CancellationToken cancels execution.

🧹 **Code Smells**

* **Info:** Uses string interpolation logging instead of structured logging placeholders.

🔧 **Refactoring Suggestions**

* Use structured logging (LogWarning("Role assignment failed {UserId} {OrgId}", …)).
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Handler contract stable under MediatR.
* Depends on IAccessService.AssignRoleAsync.

✅ **Confidence**  
High – full handler visible.

**File Review**

**File:** Application/Features/Roles/Commands/AssignRoleCommandValidator.cs  
**Layer/Type:** Application – FluentValidation Validator  
**Status:** Reviewed  
**Tokens (approx.):** ~90

🧾 **ELI5**  
This file ensures role assignment requests always include valid IDs for the user, organisation, and role.

🎯 **Purpose and Role**

* Validates AssignRoleCommand.
* Ensures IDs are not empty.

🔍 **Detailed Breakdown**  
Rules:

* UserId → must not be empty.
* OrganisationId → must not be empty.
* RoleId → must not be empty.

⚠️ **Error Handling & Validation**

* Clear error messages ("User", "Organisation", "Role").
* Rejects malformed commands early.

🔐 **Security Review**

* Safe – only validates IDs.
* No exposure of secrets.

⚡ **Performance & Reliability**

* Lightweight.
* Runs O(1).

📊 **Observability**

* Validation errors bubble through pipeline.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Empty UserId → invalid.
2. Empty OrgId → invalid.
3. Empty RoleId → invalid.
4. All valid IDs → valid.

🧹 **Code Smells**

* None – minimal and clear.

🔧 **Refactoring Suggestions**

* None required.

📜 **Contracts & Compatibility**

* Defines validity rules for AssignRole API.
* Stable contract – changes would affect clients.

✅ **Confidence**  
High – validator is small and clear.

**File Review**

**File:** Application/Features/Scopes/Queries/GetActiveScopesQuery.cs  
**Layer/Type:** Application – MediatR Query + Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~120

🧾 **ELI5**  
This file lets the system fetch all active Xero scopes. It defines the query object and its handler, which calls a scope service to retrieve the list.

🎯 **Purpose and Role**

* Implements IRequest<IReadOnlyList<XeroScope>>.
* Used to fetch currently active OAuth scopes from the configuration/persistence layer.
* Ensures only active scopes are returned for client use.

🔍 **Detailed Breakdown**

* GetActiveScopesQuery → empty query (marker).
* GetActiveScopesQueryHandler →
  + Depends on IXeroScopeService.
  + Handle method calls GetActiveScopesAsync(cancellationToken) and returns list of XeroScope.

⚠️ **Error Handling & Validation**

* No validation needed (query carries no input).
* No error handling in handler; relies on service to throw or return empty.

🔐 **Security Review**

* Only exposes scope metadata (safe).
* Must ensure service does not return inactive/invalid scopes.

⚡ **Performance & Reliability**

* Async, efficient delegation.
* DB/service call performance depends on proper indexing (ScopeName, IsActive).
* Missing try/catch may cause unhandled exceptions.

📊 **Observability**

* No logging of failures.
* Relies on IXeroScopeService for telemetry.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Service returns multiple scopes → query returns all.
2. Service returns empty → query returns empty list.
3. Service throws exception → ensure handler propagates (or could be wrapped).
4. CancellationToken cancels execution.

🧹 **Code Smells**

* **Low:** Handler lacks failure logging (inconsistent with command handlers).

🔧 **Refactoring Suggestions**

* Add try/catch with Result.Failure for consistency with other handlers.
  + Effort: Small, Priority: 3.
* Add logging of failures (warning/error).
  + Effort: Quick Win, Priority: 3.

📜 **Contracts & Compatibility**

* Defines Application query contract for active scopes.
* Stable API surface unless XeroScope changes.

✅ **Confidence**  
High – file is short, complete, and consistent with architectural intent.

**File Review**

**File:** Application/Features/Dashboard/Queries/GetDashboardQuery.cs  
**Layer/Type:** Application – MediatR Query DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This file defines the query object used to fetch the dashboard summary for a user.

🎯 **Purpose and Role**

* Represents a CQRS query (IRequest<Result<DashboardViewModel>>).
* Consumed by GetDashboardQueryHandler.
* Encapsulates the input: the user’s unique identifier.

🔍 **Detailed Breakdown**

* UserId : Guid → required user ID.
* Constructor requires UserId.

⚠️ **Error Handling & Validation**

* No inline validation.
* Relies on downstream service to validate UserId.

🔐 **Security Review**

* Only carries a UserId.
* Safe for logging.

⚡ **Performance & Reliability**

* Lightweight DTO.
* Reliability depends on service query efficiency.

📊 **Observability**

* None (correct for DTO).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid UserId → accepted.
2. Empty Guid → rejected downstream.

🧹 **Code Smells**

* None.

🔧 **Refactoring Suggestions**

* Consider adding a validator to enforce non-empty UserId.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines Application query contract for dashboard data.
* Changes ripple to handler and API.

✅ **Confidence**  
High – very small file, clear intent.

**File Review**

**File:** Application/Features/Dashboard/Queries/GetDashboardQueryHandler.cs  
**Layer/Type:** Application – MediatR Query Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~120

🧾 **ELI5**  
This file handles requests to fetch a user’s dashboard summary by calling the dashboard service.

🎯 **Purpose and Role**

* Implements IRequestHandler<GetDashboardQuery, Result<DashboardViewModel>>.
* Delegates to IDashboardService.GetDashboardDataAsync.
* Returns result wrapped in Result<DashboardViewModel>.

🔍 **Detailed Breakdown**  
Dependencies:

* IDashboardService → service providing dashboard aggregation.

Handle method:

* Calls GetDashboardDataAsync(UserId, cancellationToken).
* Returns Result.Success(vm).

⚠️ **Error Handling & Validation**

* No try/catch block.
* Relies on service to throw or return valid data.

🔐 **Security Review**

* Only processes by UserId.
* DashboardViewModel must be verified not to leak sensitive data.

⚡ **Performance & Reliability**

* Async and scalable.
* Performance depends on dashboard service efficiency (likely joins/aggregations).
* No retry or fallback logic.

📊 **Observability**

* No logging here.
* Assumes observability handled in service or middleware.

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid UserId → returns dashboard view model.
2. Service throws exception → should propagate (or wrap).
3. CancellationToken cancels execution.

🧹 **Code Smells**

* **Low:** No error handling → inconsistent with command handlers.

🔧 **Refactoring Suggestions**

* Add try/catch to return Result.Failure when service fails.
  + Effort: Small, Priority: 3.
* Add logging for failures.
  + Effort: Quick Win, Priority: 2.

📜 **Contracts & Compatibility**

* Defines query contract for dashboard API.
* Changes ripple to service and API consumers.

✅ **Confidence**  
High – handler simple, consistent with intent.

**File Review**

**File:** Application/Features/Home/Queries/GetHomeQuery.cs  
**Layer/Type:** Application – MediatR Query DTO  
**Status:** Reviewed  
**Tokens (approx.):** ~70

🧾 **ELI5**  
This file defines a query object used to fetch the data needed for a user’s home page view.

🎯 **Purpose and Role**

* CQRS query (IRequest<Result<HomeDto>>).
* Consumed by GetHomeQueryHandler.
* Encapsulates the requesting user’s unique identifier.

🔍 **Detailed Breakdown**

* UserId : Guid → required.
* Declared as record (immutable, value semantics).

⚠️ **Error Handling & Validation**

* No inline validation.
* Empty Guid could be passed, must be rejected by validator/service.

🔐 **Security Review**

* Only carries UserId.
* Safe for logging.

⚡ **Performance & Reliability**

* Very lightweight.
* Reliable as DTO.

📊 **Observability**

* None here (correct).

🧪 **Testability & Coverage**  
Suggested test cases:

1. Valid UserId → valid query.
2. Empty UserId → rejected downstream.

🧹 **Code Smells**

* None.

🔧 **Refactoring Suggestions**

* Consider adding a validator to enforce non-empty UserId.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines API contract for Home query.
* Changes ripple to handler and API consumers.

✅ **Confidence**  
High – simple, visible, immutable.

**File Review**

**File:** Application/Features/Home/Queries/GetHomeQueryHandler.cs  
**Layer/Type:** Application – MediatR Query Handler  
**Status:** Reviewed  
**Tokens (approx.):** ~350

🧾 **ELI5**  
This file builds the home page view for a user by combining organisation data, stats, and tenant connection details.

🎯 **Purpose and Role**

* Implements IRequestHandler<GetHomeQuery, Result<HomeDto>>.
* Delegates work to:
  + IOrganisationService → fetches user’s organisations.
  + ICallLogService → retrieves latest ingestion stats.
  + IXeroTokenService → checks tenant connections.
* Constructs a composite HomeDto with organisation details, connection status, scopes, and stats.

🔍 **Detailed Breakdown**  
Key logic:

* Get user’s organisations via GetUserOrganisationsAsync.
* If none → return empty HomeDto.
* Collect OrgIds and query latest stats via GetLatestStatsAsync.
* Map organisation IDs to names.
* For each organisation:
  + Check if tenant is connected via IsTenantConnectedAsync.
  + Retrieve stats and assign organisation name.
* Build TenantDetailsDto list.
* Construct HomeDto:
  + IsConnected (if any tenant connected).
  + Organisations (list of tenant details).

Error handling:

* Wraps in try/catch, returning Result.Failure on exception with message.

⚠️ **Error Handling & Validation**

* ✅ Includes try/catch → robust against unexpected errors.
* ✅ Returns empty HomeDto when user has no organisations.
* ⚠️ Returns raw exception message in failure → risk of exposing internal details.

🔐 **Security Review**

* Potential leak: raw exception messages may contain sensitive info.
* Safer to log full details internally, but return a generic failure message externally.
* Tenant connection status and scopes are not sensitive (safe to expose to authenticated user).

⚡ **Performance & Reliability**

* Multiple async service calls.
* Per-organisation loop calls IsTenantConnectedAsync individually → risk of N+1 calls.
* Could batch connection checks for performance.

📊 **Observability**

* No explicit logging here.
* Relies on exception message returns instead of logs.

🧪 **Testability & Coverage**  
Suggested test cases:

1. User with no organisations → empty HomeDto.
2. User with multiple orgs, all connected → IsConnected = true.
3. User with mixed orgs, some connected → IsConnected = true, others false.
4. Stats returned correctly per org.
5. Exception thrown by OrgService → failure result returned.

🧹 **Code Smells**

* **Medium:** Potential N+1 service call problem for tenant connection checks.
* **Medium:** Returning raw exception messages to client.
* **Low:** No logging, only failure return.

🔧 **Refactoring Suggestions**

* Replace raw exception message with generic error message.
  + Effort: Small, Priority: 4.
* Introduce logging of exception details for observability.
  + Effort: Quick Win, Priority: 3.
* Consider batching IsTenantConnectedAsync calls for performance.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Defines contract for Home API (HomeDto).
* Changes affect API consumers directly.

✅ **Confidence**  
High – full file visible and consistent.

**File Review**

**File:** Application/Interfaces/IUnitOfWork.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~220

🧾 **ELI5**  
This file defines the contract for a Unit of Work in the application. It coordinates multiple repositories under a single transaction so that changes are either all saved or all rolled back together.

🎯 **Purpose and Role**

* Represents a transactional unit that groups repository operations.
* Ensures consistency across repositories within the same DB context.
* Consumed by services that need atomic persistence of multiple aggregates.

🔍 **Detailed Breakdown**

* Extends IAsyncDisposable (clean connection disposal).
* Exposes repositories:
  + IUserRepository Users
  + ICallLogRepository CallLogs
  + IOrganisationRepository Organisations
  + IRoleRepository Roles
  + IPermissionRepository Permissions
  + IXeroWebhookEventStore XeroWebhookEvents
* Defines method:
  + Task<int> CompleteAsync(CancellationToken) → commits pending changes atomically, returns rows affected.

⚠️ **Error Handling & Validation**

* Interface only; actual error handling depends on implementation (likely Infrastructure).
* No explicit contract for rollback behaviour (assumed).

🔐 **Security Review**

* No secrets exposed.
* Must ensure implementing classes enforce transactional integrity.

⚡ **Performance & Reliability**

* Contract implies one DB transaction per unit of work.
* Proper use prevents partial writes but may create long transactions if abused.

📊 **Observability**

* No logging specified at interface level.
* Implementations should log failures and retries.

🧪 **Testability & Coverage**  
Suggested tests (on implementation):

1. Multiple repositories commit → all changes saved.
2. Exception mid-operation → rollback enforced.
3. Empty transaction → no changes.

🧹 **Code Smells**

* **Low:** Tightly couples all repository interfaces into a single contract (reduced flexibility for partial units).

🔧 **Refactoring Suggestions**

* Consider adding BeginTransaction/Commit/Rollback for finer control.
  + Effort: Medium, Priority: 3.
* Consider splitting into smaller Units of Work if not all repositories are always needed.
  + Effort: Medium, Priority: 2.

📜 **Contracts & Compatibility**

* Core persistence contract for Application → Infrastructure boundary.
* Any changes break repository/service implementations.

✅ **Confidence**  
High – well-structured, clear contract.

**File Review**

**File:** Application/Interfaces/IUnitOfWorkFactory.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~150

🧾 **ELI5**  
This file defines the contract for creating Unit of Work instances. It’s a factory that ensures each UoW is fresh, isolated, and transactional.

🎯 **Purpose and Role**

* Provides a way to create new IUnitOfWork instances.
* Encapsulates connection/transaction setup logic.
* Used by services to obtain scoped UoW for atomic operations.

🔍 **Detailed Breakdown**

* Method:
  + Task<IUnitOfWork> CreateAsync(CancellationToken cancellationToken = default) → asynchronously creates new UoW.
* Ensures each call returns a new, independent transaction scope.

⚠️ **Error Handling & Validation**

* None at interface level.
* Responsibility lies in Infrastructure implementation.

🔐 **Security Review**

* No sensitive data handled here.
* Must ensure implementation respects secure DB connection handling (Key Vault/env vars).

⚡ **Performance & Reliability**

* Factory ensures fresh connections.
* Must avoid connection leaks.
* Async contract aligns with modern DB drivers.

📊 **Observability**

* No logging here; implementation should log failures.

🧪 **Testability & Coverage**  
Suggested tests (on implementation):

1. Multiple calls → independent UoW instances.
2. Dispose UoW → connection released.
3. Cancellation token aborts creation.

🧹 **Code Smells**

* None significant.

🔧 **Refactoring Suggestions**

* Consider adding overload for non-async creation (sync contexts).
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* Defines persistence creation contract.
* Changing return type or method signature would break all consumers.

✅ **Confidence**  
High – small, clear contract.

**File Review**

**File:** Application/Interfaces/Caching/ITokenCache.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~200

🧾 **ELI5**  
This file defines the contract for caching Xero OAuth2 tokens in memory or distributed cache. It allows retrieving, setting, and removing tokens for specific users.

🎯 **Purpose and Role**

* Provides caching abstraction for OAuth tokens.
* Decouples Application services from specific caching implementations (in-memory, Redis).
* Supports per-user, thread-safe token storage.

🔍 **Detailed Breakdown**  
Methods:

* bool TryGet(Guid userId, out XeroOAuth2Token? token) → attempts to get cached token.
* void Set(Guid userId, XeroOAuth2Token token) → adds/replaces token.
* void Remove(Guid userId) → removes token.

Contract Notes:

* Outlines per-user caching.
* Explicitly designed to be implemented by MemoryTokenCache (single instance) and RedisTokenCache (multi-instance).
* Supports concurrency safety.

⚠️ **Error Handling & Validation**

* No contracts for cache eviction or expiry – depends on implementation.
* Assumes implementations handle nulls and invalid tokens gracefully.

🔐 **Security Review**

* Tokens held in memory or distributed cache.
* Sensitive: Access/refresh tokens → must never be logged.
* Secure eviction policies (expiry, rotation) must be respected.
* Must align with ADR: in-memory cache for single-node, Redis for distributed deployments.

⚡ **Performance & Reliability**

* Provides O(1) access semantics.
* Reliability depends on eviction policies and distributed cache configuration.
* Critical for token refresh performance (avoids DB round-trips).

📊 **Observability**

* Interface does not define logging; implementations should log cache misses and refreshes.

🧪 **Testability & Coverage**  
Suggested test cases (for implementations):

1. Set token → TryGet returns true.
2. Remove token → TryGet returns false.
3. Expired token removed automatically.
4. Concurrent access across multiple threads → safe.
5. Distributed deployment: Redis cache sync across nodes.

🧹 **Code Smells**

* **Medium:** Lacks async methods – may block if backing cache is distributed (e.g., Redis).
* **Low:** No contract for eviction/expiry.

🔧 **Refactoring Suggestions**

* Add async variants (Task<X?> TryGetAsync, Task SetAsync) for distributed scenarios.
  + Effort: Medium, Priority: 4.
* Consider explicit contract for eviction/expiry policies.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Core to token management.
* Changing signatures impacts both Memory and Redis cache implementations.
* Must remain backward-compatible with ADR token refresh strategy.

✅ **Confidence**  
High – interface is clear and well aligned with project ADRs.

**File Review**

**File:** Application/Interfaces/Configuration/IWebAppConfiguration.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~60

🧾 **ELI5**  
This file defines the contract for web app–specific configuration. It provides the application’s base URL, which other components use to build links for redirects, emails, and callbacks.

🎯 **Purpose and Role**

* Exposes configuration needed by Application layer services (e.g., Auth, Email).
* Abstracts away environment-specific base URL details.
* Keeps Clean Architecture by not depending directly on Infrastructure configuration.

🔍 **Detailed Breakdown**

* Property:
  + string BaseUrl { get; } → root URL for the web app (e.g., https://app.domain.com).

⚠️ **Error Handling & Validation**

* Interface only.
* Implementation must validate BaseUrl format.

🔐 **Security Review**

* No secrets here.
* Must ensure HTTPS enforced in production.

⚡ **Performance & Reliability**

* Extremely lightweight.
* Reliability depends on correct environment configuration injection.

📊 **Observability**

* None required at interface level.

🧪 **Testability & Coverage**  
Suggested tests (on implementation):

1. BaseUrl loaded correctly from configuration.
2. Invalid or missing BaseUrl → fail early.
3. Ensure HTTPS enforced for production.

🧹 **Code Smells**

* **Info:** Interface is minimal; easy to extend if more web-app–specific settings are added.

🔧 **Refactoring Suggestions**

* Consider grouping related web config (e.g., BaseUrl, FrontendUrl, SupportUrl) into a single contract.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Used by Application layer.
* Any changes ripple into services consuming BaseUrl (e.g., email templates, OAuth redirect URIs).

✅ **Confidence**  
High – very simple, clear interface.

**File Review**

**File:** Application/Interfaces/External/IEmailService.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~150

🧾 **ELI5**  
This defines the contract for sending transactional emails, such as general notifications and password reset emails.

🎯 **Purpose and Role**

* Abstracts email sending logic.
* Allows Application layer to request email delivery without depending on SMTP/provider details.

🔍 **Detailed Breakdown**  
Methods:

* Task<Result<bool>> SendEmailAsync(MailMessage message, CancellationToken token = default)
* Task<Result<bool>> SendPasswordResetEmailAsync(string email, string resetToken, string callbackUrl, CancellationToken token = default)

⚠️ **Error Handling & Validation**

* Returns Result<bool> to indicate success/failure.
* Implementation must handle invalid email formats and SMTP errors.

🔐 **Security Review**

* Password reset email method carries sensitive reset token → must only send hashed or one-time-use token.
* No logging of tokens allowed.

⚡ **Performance & Reliability**

* Async contract allows non-blocking sending.
* Reliability depends on provider’s SLA and retry policies.

📊 **Observability**

* Implementation should log failures with correlation IDs.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid MailMessage → success.
2. Invalid address → failure.
3. SMTP error → failure result.
4. Password reset email → includes reset link.

🧹 **Code Smells**

* **Info:** Password reset logic is tied to this interface; could be factored into separate AuthEmailService.

🔧 **Refactoring Suggestions**

* Separate transactional email vs security-sensitive (password reset).
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Core Application contract.
* Changing signatures impacts Infrastructure email implementations.

✅ **Confidence**  
High – clear, aligned.

**File Review**

**File:** Application/Interfaces/External/IXeroApiClient.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~400

🧾 **ELI5**  
This defines the contract for interacting with the Xero API. It covers token management, tenant connections, data fetching, and login flow.

🎯 **Purpose and Role**

* Encapsulates all Xero API calls.
* Keeps Application services abstracted from SDK details.

🔍 **Detailed Breakdown**  
Key methods:

* **Connections:**
  + GetConnectionsAsync → retrieves connected tenants.
  + RefreshTokenAsync → exchanges refresh for new access/refresh.
  + DeleteConnectionAsync → revokes tenant connection.
* **Login flow:**
  + BuildLoginUri(state, scopes, redirectUri) → returns OAuth2 login URL.
  + GetAccessTokenAsync(code, redirectUri) → exchanges code for tokens.
* **Data ingestion:**
  + FetchDataForEndpointAsync → retrieves data for a specific endpoint (paged).
  + GetInvoiceAsync → fetches raw invoice JSON.

⚠️ **Error Handling & Validation**

* Returns DTOs (XeroRefreshResponseDto, EndpointIngestionResult, etc.) or null.
* Must handle API rate limits, expired tokens, and revoked tenants.

🔐 **Security Review**

* Handles access and refresh tokens → critical to ensure no logging of tokens.
* Must enforce secure storage (Key Vault/env vars).
* Tenant isolation must be maintained.

⚡ **Performance & Reliability**

* Network-bound operations → must implement retries with exponential backoff.
* Paging ensures large data ingestion manageable.
* Rate limiting must be respected (Xero’s API rules).

📊 **Observability**

* Implementations should log request URIs, tenant IDs, response codes (never tokens).
* Metrics for rate limit consumption recommended.

🧪 **Testability & Coverage**  
Suggested tests (integration):

1. Valid token → fetch connections.
2. Expired token → refresh successful.
3. Revoked tenant → delete connection handled.
4. Large dataset → fetch multiple pages.

🧹 **Code Smells**

* **Medium:** Very large interface → SRP violation. Combines token management, login, and data ingestion.

🔧 **Refactoring Suggestions**

* Split into sub-interfaces: IXeroAuthClient, IXeroDataClient.
  + Effort: Large, Priority: 3.

📜 **Contracts & Compatibility**

* Critical Application–Infrastructure contract.
* Any change requires Infrastructure + service refactor.

✅ **Confidence**  
High – clear mapping to ADR for Xero integration.

**File Review**

**File:** Application/Interfaces/External/IXeroRawDataSynchronizer.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~130

🧾 **ELI5**  
This defines the contract for synchronizing raw Xero data (ETL). It fetches endpoint data for a user/tenant and stores it.

🎯 **Purpose and Role**

* Abstracts ingestion of raw Xero API responses.
* Provides Application with batch ETL logic without exposing implementation details.

🔍 **Detailed Breakdown**  
Method:

* Task<IReadOnlyList<IngestionReport>> RunOnceAsync(Guid userId, Guid tenantId, CancellationToken token = default, string? endpointKey = null)

⚠️ **Error Handling & Validation**

* Must handle invalid user/tenant IDs.
* Must enforce endpointKey validity.

🔐 **Security Review**

* Works with user/tenant token context.
* Must not log sensitive token data.

⚡ **Performance & Reliability**

* ETL jobs may be long-running.
* Should support cancellation tokens for safe termination.
* Parallelisation strategies may be required.

📊 **Observability**

* Implementation should log ingestion progress and errors per endpoint.

🧪 **Testability & Coverage**  
Suggested tests:

1. Run once with valid inputs → returns reports.
2. Invalid tenantId → failure.
3. EndpointKey = null → processes all endpoints.
4. Cancel token → job halts gracefully.

🧹 **Code Smells**

* **Low:** Only a single method; limited extensibility if sync modes increase.

🔧 **Refactoring Suggestions**

* Consider adding batch vs single-endpoint methods explicitly.
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* Defines ingestion ETL contract.
* Any change impacts Infrastructure ETL pipeline.

✅ **Confidence**  
High – concise, clear contract.

**File Review**

**File:** Application/Interfaces/Locking/IDistributedLock.cs  
**Layer/Type:** Application – Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~90

🧾 **ELI5**  
This defines the contract for a distributed lock. It ensures that only one process or instance performs a critical operation at a time, even in multi-server deployments.

🎯 **Purpose and Role**

* Provides concurrency control across distributed systems.
* Used for token refresh, background jobs, and multi-instance safe operations.
* Abstracts away implementation (e.g., Redis-based lock).

🔍 **Detailed Breakdown**  
Methods:

* Task<bool> AcquireAsync(string key, TimeSpan expiry, CancellationToken token = default) → attempts to acquire a lock for a key, returns true/false.
* Task ReleaseAsync(string key) → releases the lock for the key.

⚠️ **Error Handling & Validation**

* Contract does not define behaviour for expired/abandoned locks.
* Implementations must ensure safety against deadlocks and stale locks.

🔐 **Security Review**

* Lock keys must not contain sensitive data (e.g., raw tokens).
* Safe to log lock acquisition/release events.

⚡ **Performance & Reliability**

* Distributed lock implementations (e.g., Redis) must be performant under load.
* Expiry prevents deadlocks but must be carefully tuned.
* Must handle network partitions gracefully.

📊 **Observability**

* Implementations should log lock contention, timeouts, and failures.
* Metrics: lock acquisition rate, contention rate.

🧪 **Testability & Coverage**  
Suggested tests:

1. Acquire lock → success.
2. Acquire same lock from another instance → fail.
3. Release lock → second instance can acquire.
4. Expired lock → second instance can acquire after expiry.
5. Concurrent acquisitions across multiple threads.

🧹 **Code Smells**

* **Info:** Interface doesn’t support re-entrant or hierarchical locks.

🔧 **Refactoring Suggestions**

* Consider adding async disposal (IAsyncDisposable) pattern.
  + Effort: Small, Priority: 2.
* Consider contract for RenewAsync to extend lock duration.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Critical for token refresh ADR (ensures only one refresh at a time per user).
* Changing method signatures impacts Infrastructure (Redis/in-memory lock).

✅ **Confidence**  
High – clear, minimal contract consistent with ADRs.

**File Review**

**File:** Application/Interfaces/Repositories/ICallLogRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~250

🧾 **ELI5**  
This defines the contract for saving and retrieving API call logs. It tracks requests to Xero endpoints (success, failure, rows inserted, errors) and provides stats.

🎯 **Purpose and Role**

* Abstracts persistence for API call logs.
* Enables Application services to record API interactions for auditing and monitoring.
* Provides read methods for dashboard/stats features.

🔍 **Detailed Breakdown**  
Key methods:

* **Write Logs**
  + LogCallAsync(callTime, organisationId, userId?, endpoint, statusCode, isSuccess, rowsInserted, errorMessage?, cancellationToken) → inserts a log entry.
* **Read Logs**
  + GetLogsAsync(organisationId, cancellationToken) → retrieves logs for an organisation.
  + GetLatestStatsAsync(callGroupIds, cancellationToken) → retrieves aggregated stats for orgs.

DTOs:

* Uses ApiCallLogEntry (individual call log).
* Uses CallStats (aggregated stats dictionary).

⚠️ **Error Handling & Validation**

* Interface only; no explicit validation.
* Implementation must:
  + Prevent invalid status codes.
  + Enforce non-null endpoint.

🔐 **Security Review**

* Logs may contain error messages → risk of exposing sensitive tokens or payloads.
* Must sanitize errorMessage field.

⚡ **Performance & Reliability**

* Logging is append-only → performant if indexed.
* GetLogsAsync must use pagination to avoid large responses.
* GetLatestStatsAsync should aggregate efficiently (indexes on OrgId, CallTime).

📊 **Observability**

* Repository itself stores observability data.
* Implementation should ensure time-based indexes for efficient queries.

🧪 **Testability & Coverage**  
Suggested tests:

1. Log call success → entry persisted.
2. Log call failure with error → entry includes errorMessage.
3. GetLogsAsync returns only for specified organisation.
4. GetLatestStatsAsync aggregates correctly across multiple orgs.
5. CancellationToken stops query.

🧹 **Code Smells**

* **Medium:** errorMessage is free-form string; risks unstructured log data.
* **Low:** No contract for pagination in GetLogsAsync.

🔧 **Refactoring Suggestions**

* Introduce structured error codes instead of free-form messages.
  + Effort: Medium, Priority: 4.
* Add paging support to GetLogsAsync.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Core auditing contract for API calls.
* Changing method signatures impacts dashboard and logging services.

✅ **Confidence**  
High – clear interface, aligned with logging and observability strategy.

**File Review**

**File:** Application/Interfaces/Repositories/IEndpointConfigRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~200

🧾 **ELI5**  
This defines the contract for managing endpoint configuration records. These configs determine which Xero endpoints are available and how they’re set up.

🎯 **Purpose and Role**

* Abstracts persistence of EndpointConfig entities.
* Provides CRUD + upsert operations.
* Allows Application layer to retrieve endpoint configs by ID or name.

🔍 **Detailed Breakdown**  
Methods:

* **Read:**
  + GetAllAsync() → all configs.
  + GetByIdAsync(int endpointId) → single config by ID.
  + GetByNameAsync(string name) → config by logical name.
* **Write:**
  + AddAsync(config) → insert new config, returns new ID.
  + UpdateAsync(config) → update existing config, returns bool.
  + UpsertAsync(config) → insert or update based on name, returns ID.
  + DeleteAsync(string name) → deletes config by name.

⚠️ **Error Handling & Validation**

* No explicit error contracts.
* Implementations must:
  + Ensure unique names.
  + Validate config before persistence.

🔐 **Security Review**

* Endpoint configs may contain sensitive details (URIs, scopes).
* Must not expose secrets via logs.
* DB must enforce appropriate constraints.

⚡ **Performance & Reliability**

* UpsertAsync can cause contention if high-frequency updates → needs index on name.
* Reads are straightforward but should be cached where possible.

📊 **Observability**

* No logging specified at interface level.
* Implementations should log mutations (add/update/delete).

🧪 **Testability & Coverage**  
Suggested tests:

1. Add new endpoint → retrievable via name.
2. Update existing → updated values persisted.
3. Upsert with existing name → replaces config.
4. Delete non-existent endpoint → returns false.
5. Concurrent upserts → ensure consistency.

🧹 **Code Smells**

* **Low:** Interface mixes query + mutation + upsert responsibilities.

🔧 **Refactoring Suggestions**

* Consider separating read and write responsibilities (CQRS repository split).
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Contract defines key persistence API for endpoint configs.
* Changes break ingestion and dashboard features relying on endpoint configs.

✅ **Confidence**  
High – interface is well-defined, consistent with clean repo pattern.

**File Review**

**File:** Application/Interfaces/Repositories/IOrganisationRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~350

🧾 **ELI5**  
This defines the contract for managing organisations (tenants), including CRUD operations, user assignments, and user-organisation-role links.

🎯 **Purpose and Role**

* Provides persistence abstraction for Organisation entities.
* Supports tenant-level queries and updates.
* Manages relationships between users, organisations, and roles.

🔍 **Detailed Breakdown**  
Key methods:

* **Read:**
  + FindByIdAsync(Guid organisationId) → returns organisation or null.
  + GetUserCountAsync(Guid organisationId) → number of users in org.
  + GetUserOrganisationsAsync(Guid userId) → list of orgs for a user.
* **Write:**
  + UpdateOrganisationAsync(Organisation org) → updates organisation fields.
  + AssignUsersAsync(Guid orgId, List<Guid> userIds) → associates users to org.
  + EnsureUserOrganisationLinkAsync(Guid orgId, Guid userId, Guid roleId, string scopes) → ensures user/org/role link exists.
* **Specialised:**
  + EnsureUserOrganisationLinkAsync performs *upsert-style* association of user-org-role.
  + GetUserOrganisationsAsync returns UserOrganisationDto (rich mapping).

⚠️ **Error Handling & Validation**

* No explicit contract for duplicate assignment or invalid org IDs.
* Must ensure transactional consistency:
  + e.g., assigning users + roles in the same transaction.

🔐 **Security Review**

* Role assignment is security-sensitive.
* Repository must enforce that scopes are validated upstream.
* Prevent privilege escalation through incorrect repository usage.

⚡ **Performance & Reliability**

* GetUserOrganisationsAsync could involve multiple joins → must be indexed on UserId + OrgId.
* Bulk operations (AssignUsersAsync) should be batched.
* Ensure idempotency of EnsureUserOrganisationLinkAsync.

📊 **Observability**

* No logging at contract level.
* Implementations should log user-org-role changes for audit.

🧪 **Testability & Coverage**  
Suggested tests:

1. FindById returns null for unknown org.
2. User count matches number of assignments.
3. AssignUsersAsync adds multiple users to org.
4. EnsureUserOrganisationLink prevents duplicates.
5. User-org-role mapping round-trips correctly.

🧹 **Code Smells**

* **Medium:** Interface mixes CRUD and relationship management (SRP concern).
* **Low:** Scope handling is string-based, not strongly typed.

🔧 **Refactoring Suggestions**

* Consider splitting into IOrganisationRepository and IUserOrganisationRepository.
  + Effort: Medium, Priority: 4.
* Replace string scopes with strongly typed value objects.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Core to multi-tenant architecture.
* Changes ripple across user management and auth services.

✅ **Confidence**  
High – interface comprehensive, well-aligned with ADRs.

**File Review**

**File:** Application/Interfaces/Repositories/IPasswordResetTokenRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~170

🧾 **ELI5**  
This defines the contract for storing, retrieving, and managing password reset tokens.

🎯 **Purpose and Role**

* Abstracts persistence for PasswordResetToken entities.
* Used by password reset flows (forgot password, reset link).
* Ensures one-time-use, time-limited reset tokens can be managed securely.

🔍 **Detailed Breakdown**  
Methods:

* FindAsync(Guid tokenId) → retrieves token by ID.
* AddAsync(PasswordResetToken token) → inserts new reset token.
* MarkAsUsedAsync(Guid tokenId) → marks token as consumed.

⚠️ **Error Handling & Validation**

* No explicit guarantees on expired tokens.
* Must ensure:
  + Only valid tokens can be used.
  + Expired tokens are rejected.

🔐 **Security Review**

* Handles sensitive reset tokens → must store only **hashed tokens**.
* No logging of token IDs.
* MarkAsUsedAsync prevents replay → critical for security.

⚡ **Performance & Reliability**

* Operations are lightweight.
* Requires index on tokenId and ExpiresAt.
* Must handle cleanup of expired tokens.

📊 **Observability**

* Implementations should log token creation and usage (without exposing token).

🧪 **Testability & Coverage**  
Suggested tests:

1. Add valid token → retrievable.
2. Expired token → cannot be used.
3. MarkAsUsed prevents reuse.
4. Invalid ID → returns null.

🧹 **Code Smells**

* **Medium:** No explicit expiry handling in contract.
* **Low:** Only tokenId lookup supported; no query by UserId.

🔧 **Refactoring Suggestions**

* Add method: RemoveExpiredTokensAsync().
  + Effort: Small, Priority: 4.
* Add query by UserId for auditing.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Key for authentication lifecycle.
* Changes ripple through Auth services.

✅ **Confidence**  
High – clear, well-aligned with password reset ADR.

**File Review**

**File:** Application/Interfaces/Repositories/IPermissionRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~160

🧾 **ELI5**  
This defines the contract for checking whether a user has certain permissions within an organisation, such as admin rights or endpoint access.

🎯 **Purpose and Role**

* Abstracts permission checks from Infrastructure.
* Centralises authorisation logic at repository level.
* Supports both role-based and endpoint-specific permission checks.

🔍 **Detailed Breakdown**  
Key methods:

* IsPlatformAdminAsync(Guid userId) → true if user is a platform admin.
* CanManageOrganisationAsync(Guid userId, Guid organisationId) → checks if user can administer an organisation.
* CanTriggerEndpointAsync(Guid userId, Guid organisationId) → checks if user can trigger specific endpoint ingestion.

⚠️ **Error Handling & Validation**

* Returns bool wrapped in Task.
* No explicit error handling – assumes user/org IDs are valid.
* Must handle missing user or org gracefully.

🔐 **Security Review**

* Critical for access control.
* Implementations must ensure permissions are derived from secure role/assignment tables.
* Risk: incorrect implementation → privilege escalation.

⚡ **Performance & Reliability**

* Permission checks likely frequent.
* Must be optimised with DB indexes on UserId + OrgId.
* Potential caching layer useful for repetitive checks.

📊 **Observability**

* Implementations should log failed access attempts.
* Important for audit trail.

🧪 **Testability & Coverage**  
Suggested tests:

1. Admin user → returns true.
2. Normal user without role → returns false.
3. Org admin → can manage their org, not others.
4. Missing user/org → returns false.
5. Concurrent checks → consistent results.

🧹 **Code Smells**

* **Medium:** All checks are boolean-returning methods; lacks granularity (e.g., permission list).
* **Low:** No way to extend with custom permission types without changing contract.

🔧 **Refactoring Suggestions**

* Introduce method GetUserPermissionsAsync(userId, orgId) returning collection.
  + Effort: Medium, Priority: 4.
* Add structured audit logging integration.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Central Application contract for authorisation.
* Any changes ripple across all access control logic.

✅ **Confidence**  
High – simple, clear, but must be implemented with extreme care for security.

**File Review**

**File:** Application/Interfaces/Repositories/IPollingSettingsRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~230

🧾 **ELI5**  
This defines the contract for managing polling settings (schedules and frequencies) that determine how often background jobs like ingestion run.

🎯 **Purpose and Role**

* Abstracts persistence of PollingSetting entities.
* Supports CRUD and bulk retrieval operations.
* Used by polling services to manage job schedules per organisation.

🔍 **Detailed Breakdown**  
Key methods:

* **Read:**
  + GetAsync(Guid organisationId) → polling settings for org.
  + GetManyAsync(List<Guid> organisationIds) → polling settings for multiple orgs.
  + GetAllAsync() → settings for all orgs.
* **Write:**
  + UpsertAsync(Guid orgId, PollingFrequency freq, TimeSpan? runTime) → create/update org schedule.

⚠️ **Error Handling & Validation**

* No explicit validation of runTime vs frequency.
* Must enforce consistency:
  + Hourly → runTime should be null.
  + Daily → runTime required.

🔐 **Security Review**

* Settings are not sensitive themselves.
* Risk only if malicious input causes mis-scheduling (DoS vector).

⚡ **Performance & Reliability**

* Bulk retrieval (GetManyAsync, GetAllAsync) useful for scheduler startup.
* Index on OrganisationId critical.
* Upsert operations must be idempotent.

📊 **Observability**

* Implementations should log schedule updates.
* Useful for auditing job scheduling changes.

🧪 **Testability & Coverage**  
Suggested tests:

1. New org → Upsert creates record.
2. Existing org → Upsert updates record.
3. Hourly + runTime provided → reject.
4. Daily + runTime missing → reject.
5. Bulk retrieval returns correct mapping.

🧹 **Code Smells**

* **Medium:** Contract doesn’t encode frequency/runTime rules.
* **Low:** No way to delete settings explicitly (only overwrite).

🔧 **Refactoring Suggestions**

* Add DeleteAsync(Guid orgId).
  + Effort: Small, Priority: 3.
* Introduce validation method to enforce rules.
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Central contract for background job scheduling.
* Changes impact polling services and dashboard features.

✅ **Confidence**  
High – clear interface, aligned with background job ADR.

**File Review**

**File:** Application/Interfaces/Repositories/IRawXeroPayloadRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~350

🧾 **ELI5**  
This defines the contract for storing raw JSON payloads fetched from Xero. It supports bulk inserts, staged ingestion, and retrieval for auditing or retrying.

🎯 **Purpose and Role**

* Provides persistence abstraction for raw Xero API payloads.
* Supports ingestion jobs (ETL) by persisting raw JSON before processing.
* Ensures resiliency: raw data available for replay/reprocessing.

🔍 **Detailed Breakdown**  
Key methods:

* **Insert:**
  + InsertPageAsync(tableName, pageNumber, payloadJson, tenantId, sourceName) → insert one ingestion page.
  + InsertShreddedRecordsAsync(tableName, payloadJson, tenantId, pageNumber, responseKey, sourceName, externalEntityIdSource?) → insert multiple JSON rows.
* **Retrieve:**
  + GetLastFetchedAsync(tableName, tenantId, sourceName) → timestamp of last successful fetch.
  + GetMaxPageNumberAsync(schemaTableName, tenantId, sourceName) → highest page number fetched.
* **Migrate:**
  + MergeStagingAsync(stagingTable, tenantId) → move staged rows into ingestion table.

⚠️ **Error Handling & Validation**

* No explicit handling of invalid JSON.
* Must enforce:
  + Payloads are valid JSON.
  + Prevent duplicate ingestion (idempotency).

🔐 **Security Review**

* Payloads may include sensitive financial/PII.
* Must ensure:
  + Encrypted at rest (DB column or table-level).
  + Sanitisation before exposing in logs.

⚡ **Performance & Reliability**

* Bulk inserts must be optimised (batch SQL).
* Merge staging must be transactional to prevent data loss.
* Indexes on TenantId, PageNumber, SourceName required.

📊 **Observability**

* Implementations should log:
  + Payload insert success/failure.
  + Merge completion with row counts.

🧪 **Testability & Coverage**  
Suggested tests:

1. Insert valid JSON → retrievable.
2. Insert invalid JSON → rejected.
3. Re-insert same page number → ignored/overwritten as per policy.
4. GetLastFetched returns null when no data.
5. Merge moves all staged rows correctly.

🧹 **Code Smells**

* **Medium:** Heavy reliance on string tableName → risk of SQL injection if unchecked.
* **Low:** Contract mixes ingestion and staging responsibilities.

🔧 **Refactoring Suggestions**

* Replace string tableName with strongly typed enum/value object.
  + Effort: Medium, Priority: 4.
* Separate ingestion vs staging repository contracts.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Central for ETL pipeline.
* Changes ripple through ingestion services and dashboards.

✅ **Confidence**  
High – interface is comprehensive and critical to ingestion ADRs.

**File Review**

**File:** Application/Interfaces/Repositories/IRoleRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This defines the contract for retrieving the default application role from the database.

🎯 **Purpose and Role**

* Abstracts persistence of RoadmappRole.
* Ensures Application services can query the default role without knowing DB details.
* Used during user creation and onboarding flows.

🔍 **Detailed Breakdown**

* Task<Result<RoadmappRole?>> GetDefaultAsync(CancellationToken token) → retrieves default role, returns null if none exists.

⚠️ **Error Handling & Validation**

* Returns Result<T> wrapper for success/failure.
* If no default exists, returns null.
* Implementation must handle case where multiple defaults exist (DB integrity issue).

🔐 **Security Review**

* Roles control access → misconfigured defaults risk privilege escalation.
* Repository must enforce unique default role constraint in DB.

⚡ **Performance & Reliability**

* Simple read query.
* Should be cached (default role rarely changes).

📊 **Observability**

* Log if default role missing.
* Important for diagnosing onboarding failures.

🧪 **Testability & Coverage**  
Suggested tests:

1. Default role exists → retrieved.
2. No default role → returns null.
3. Multiple defaults (invalid DB) → returns error.

🧹 **Code Smells**

* **Low:** Very narrow contract – might be better folded into a general role repository.

🔧 **Refactoring Suggestions**

* Expand to include other role queries (by name, by ID).
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Critical for default onboarding role.
* Breaking changes impact user creation flows.

✅ **Confidence**  
High – small, clear, security-sensitive contract.

**File Review**

**File:** Application/Interfaces/Repositories/ITokenRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~450

🧾 **ELI5**  
This defines the contract for securely storing, retrieving, updating, and deleting OAuth tokens (access + refresh) used to connect to Xero.

🎯 **Purpose and Role**

* Abstracts persistence of TokenRecord entities.
* Provides CRUD operations for access/refresh tokens.
* Supports multi-tenant architecture by associating tokens with users and tenants.
* Ensures token lifecycle (insert, update, delete, expire).

🔍 **Detailed Breakdown**  
Key methods:

* **Read:**
  + GetTokensForUserAsync(Guid userId) → returns all tokens for a user.
  + GetAllUserIdsAsync() → all users with tokens.
  + GetUsersWithStaleTokensAsync(int olderThanDays) → users whose tokens expired.
* **Write:**
  + StoreTokensAsync(TokenRecord record) → insert or update.
  + StoreTokenAsync(TokenRecord record) → single save (legacy/obsolete).
  + DeleteTokensForUserAsync(Guid userId) → remove all user’s tokens.
  + DeleteTokenForTenantAsync(Guid userId, Guid tenantId) → remove specific tenant link.
* **Metadata:**
  + MarkAsUsedAsync(Guid tokenId) → mark refresh token as consumed.
  + GetUsersWithStaleTokensAsync → supports rotation cleanup.

⚠️ **Error Handling & Validation**

* No explicit validation in contract.
* Implementation must:
  + Enforce uniqueness per (UserId, TenantId).
  + Handle race conditions (simultaneous refresh attempts).

🔐 **Security Review**

* Handles sensitive OAuth tokens:
  + Must encrypt at rest (DB or Key Vault).
  + Must never log token values.
* GetUsersWithStaleTokensAsync → critical for automated cleanup.

⚡ **Performance & Reliability**

* Token reads/writes must be low latency (used in auth flows).
* Bulk deletes (DeleteTokensForUserAsync) must cascade efficiently.
* Indexes required: UserId, TenantId, Expiry.

📊 **Observability**

* Implementations should log token creation/rotation (never token values).
* Failed refreshes must be logged for audit.

🧪 **Testability & Coverage**  
Suggested tests:

1. Store + retrieve token → values consistent.
2. Expired tokens → retrieved by GetUsersWithStaleTokens.
3. Delete for tenant removes only targeted entry.
4. Concurrent refresh attempts → one succeeds, others fail.
5. MarkAsUsed ensures refresh token not reused.

🧹 **Code Smells**

* **Medium:** Interface is very large, combining CRUD, lifecycle, and cleanup responsibilities.
* **Low:** Overlap between StoreTokensAsync and StoreTokenAsync → duplication.

🔧 **Refactoring Suggestions**

* Remove obsolete StoreTokenAsync, keep unified StoreTokensAsync.
  + Effort: Small, Priority: 4.
* Split into sub-interfaces:
  + ITokenStore, ITokenCleaner.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Central to authentication and token lifecycle ADR.
* Any breaking changes require refactor of Auth services and background refresh jobs.

✅ **Confidence**  
High – interface is well aligned with OAuth token management but too large.

**File Review**

**File:** Application/Interfaces/Repositories/IUserRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~450

🧾 **ELI5**  
This defines the contract for managing users, including CRUD, authentication details (password hash, Xero user link), organisation memberships, and roles.

🎯 **Purpose and Role**

* Provides persistence abstraction for RoadmappUser.
* Supports login, registration, password management, and user-org-role assignments.
* Used by auth services, organisation features, and role management.

🔍 **Detailed Breakdown**  
Key methods:

* **Find users:**
  + FindByUsernameAsync(username)
  + FindByEmailAsync(email)
  + FindByXeroUserIdAsync(xeroUserId)
  + FindByIdAsync(userId)
* **Check existence:**
  + ExistsAsync(username, email) → returns bool.
* **Add/update:**
  + AddAsync(user, passwordHash) → creates user with hash.
  + UpdatePasswordAsync(userId, newPasswordHash) → password change.
  + UpdateProfileAsync(user) → update display fields.
  + SetXeroUserIdAsync(userId, xeroUserId) → links Roadmapp and Xero users.
* **Org/role management:**
  + GetUserOrgNamesAsync(userId) → organisation names for user.
  + GetUserRolesAsync(userId) → roles within orgs.
  + AssignRoleAsync(userId, orgId, roleId) → set/change user’s role.
  + RemoveUserOrgAccessAsync(userId, orgId) → remove access.

⚠️ **Error Handling & Validation**

* Contract leaves validation to services.
* Risks:
  + Duplicate usernames/emails not prevented here.
  + AssignRoleAsync must validate role existence.

🔐 **Security Review**

* Handles password hashes → must be securely hashed (BCrypt/Argon2).
* No plaintext passwords should touch repo.
* Linking to Xero IDs → risk of spoofing unless verified.

⚡ **Performance & Reliability**

* Must index Username, Email, UserId for fast lookup.
* Assign/remove roles must be transactional with org link updates.
* Bulk queries (GetUserRolesAsync) may be heavy → caching beneficial.

📊 **Observability**

* Should log:
  + Failed login lookups.
  + Role/Org assignment changes.

🧪 **Testability & Coverage**  
Suggested tests:

1. FindByUsername returns correct user.
2. Duplicate email → ExistsAsync returns true.
3. Add user persists hashed password.
4. AssignRole updates user-org-role mapping.
5. RemoveUserOrgAccess prevents access.

🧹 **Code Smells**

* **High:** Interface is very large → mixes auth, profile, org membership, and role management.
* **Medium:** Password and profile updates in same repo risk violating SRP.

🔧 **Refactoring Suggestions**

* Split into:
  + IUserRepository (CRUD + auth).
  + IUserOrgRepository (org memberships).
  + IUserRoleRepository (roles).
  + Effort: Large, Priority: 3.

📜 **Contracts & Compatibility**

* Central to authentication and user management.
* Breaking changes cascade into most of Application layer.

✅ **Confidence**  
High – contract is clear but overloaded.

**File Review**

**File:** Application/Interfaces/Repositories/IXeroScopeRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~200

🧾 **ELI5**  
This defines the contract for storing and retrieving Xero OAuth scopes (e.g., accounting.transactions).

🎯 **Purpose and Role**

* Abstracts persistence of XeroScope entities.
* Provides CRUD-like methods for scopes.
* Used to check active scopes and support OAuth token handling.

🔍 **Detailed Breakdown**  
Key methods:

* **Read:**
  + GetAllAsync() → all scopes.
  + GetActiveScopesAsync() → only active scopes.
  + GetByIdAsync(id) → scope by ID.
  + GetByScopeNameAsync(scopeName) → scope by string name.

⚠️ **Error Handling & Validation**

* Returns null if scope not found.
* Must enforce uniqueness on scope name.

🔐 **Security Review**

* Scopes are not secrets themselves.
* But incorrect configuration may grant excessive permissions → must align with ADR (scopes as collections, not free-form strings).

⚡ **Performance & Reliability**

* Lookups should be indexed on ID and Name.
* Active scopes query must be efficient.

📊 **Observability**

* Log if scope missing when expected.
* Important for diagnosing OAuth failures.

🧪 **Testability & Coverage**  
Suggested tests:

1. Insert + retrieve scope by name.
2. Inactive scope not included in active list.
3. Duplicate scope name prevented.
4. GetById returns null for unknown ID.

🧹 **Code Smells**

* **Medium:** Scope name stored as string → should use enum/value object.

🔧 **Refactoring Suggestions**

* Replace raw string scope names with typed object.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Key for OAuth scope management.
* Changes ripple through auth flows.

✅ **Confidence**  
High – interface is small, clear, but relies on string-based scopes.

**File Review**

**File:** Application/Interfaces/Repositories/IXeroWebhookEventStore.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~120

🧾 **ELI5**  
This defines the contract for persisting webhook events from Xero so they can be processed exactly once.

🎯 **Purpose and Role**

* Stores incoming webhook events.
* Ensures idempotency by checking if an event has already been processed.
* Supports marking events as processed once handled.

🔍 **Detailed Breakdown**  
Methods:

* HasBeenProcessedAsync(Guid eventId) → checks if event already handled.
* MarkAsProcessedAsync(XeroWebhookEvent event) → saves/marks event as processed.

⚠️ **Error Handling & Validation**

* Contract does not specify behaviour for duplicate event IDs.
* Must handle race conditions → two services trying to process same event.

🔐 **Security Review**

* Webhook data may contain sensitive financial info.
* Must validate webhook signature before calling repository.
* Must avoid logging raw webhook payload.

⚡ **Performance & Reliability**

* Idempotency critical → requires unique index on EventId.
* Must scale to handle burst traffic from Xero webhooks.

📊 **Observability**

* Implementations should log rejected duplicates and processing errors.

🧪 **Testability & Coverage**  
Suggested tests:

1. Insert new event → returns not processed.
2. Duplicate event ID → HasBeenProcessed true.
3. MarkAsProcessed persists event.
4. Concurrency: two inserts, only one succeeds.

🧹 **Code Smells**

* **Low:** Only stores processed events; no contract for failed/retry state.

🔧 **Refactoring Suggestions**

* Extend contract to include retry/failure handling.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Central to webhook ingestion reliability.
* Changes impact webhook services directly.

✅ **Confidence**  
High – clear and aligned with idempotent processing ADR.

**File Review**

**File:** Application/Interfaces/Repositories/IXeroWebhookHistoryRepository.cs  
**Layer/Type:** Application – Repository Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~50

🧾 **ELI5**  
This defines the contract for storing raw webhook payloads from Xero for historical/audit purposes.

🎯 **Purpose and Role**

* Provides persistence abstraction for webhook payloads.
* Supports auditing, debugging, and replay of webhook data.

🔍 **Detailed Breakdown**

* Method:
  + SaveAsync(string payloadJson) → saves raw webhook payload as string.

⚠️ **Error Handling & Validation**

* No contract for invalid JSON.
* Implementation must handle:
  + Large payloads.
  + Invalid or truncated JSON.

🔐 **Security Review**

* Webhook payloads may contain sensitive financial info.
* Must encrypt or store securely.
* Logs must not output raw payload.

⚡ **Performance & Reliability**

* Inserts must be efficient → may require append-only logging table.
* Payload size could be large → must handle safely.

📊 **Observability**

* Should log payload metadata (ID, timestamp), not full JSON.

🧪 **Testability & Coverage**  
Suggested tests:

1. Save valid payload → retrievable.
2. Save invalid payload → rejected.
3. Save large payload → stored successfully.

🧹 **Code Smells**

* **Low:** Only supports insert; no retrieval contract defined.

🔧 **Refactoring Suggestions**

* Add retrieval method for audit queries.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Supports webhook auditing.
* Changes ripple to monitoring/audit services.

✅ **Confidence**  
High – simple, focused contract.

**File Review**

**File:** Application/Interfaces/Security/IPasswordHasher.cs  
**Layer/Type:** Application – Security Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This defines the contract for hashing and verifying passwords securely, so plaintext passwords are never stored.

🎯 **Purpose and Role**

* Provides abstraction for password hashing.
* Used by auth services when creating and validating users.
* Keeps hashing implementation (BCrypt, Argon2, PBKDF2) out of Application layer.

🔍 **Detailed Breakdown**  
Methods:

* HashPasswordAsync(string password) → returns hashed string.
* VerifyPasswordAsync(string hashedPassword, string password) → returns bool (match or not).

⚠️ **Error Handling & Validation**

* Contract assumes non-null, non-empty input.
* Implementation must reject weak/short passwords before hashing.

🔐 **Security Review**

* Hash must be salted, use strong algorithm (Argon2id preferred).
* Hashing must be slow (resists brute force).
* Must never log passwords or hashes.

⚡ **Performance & Reliability**

* High CPU cost by design.
* Must balance security with login throughput.

📊 **Observability**

* Should log failed verifications (without input values).

🧪 **Testability & Coverage**  
Suggested tests:

1. Hash + verify → true.
2. Wrong password → false.
3. Different hashes for same password (salted).
4. Empty/null input → throws error.

🧹 **Code Smells**

* None at interface level.

🔧 **Refactoring Suggestions**

* Consider async disposal pattern if external libraries require cleanup.

📜 **Contracts & Compatibility**

* Core auth contract.
* Breaking changes ripple into UserRepository.

✅ **Confidence**  
High – essential, well-abstracted.

**File Review**

**File:** Application/Interfaces/Security/ITokenEncryptionService.cs  
**Layer/Type:** Application – Security Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~120

🧾 **ELI5**  
This defines the contract for encrypting and decrypting tokens (e.g., OAuth refresh tokens) so they’re secure at rest.

🎯 **Purpose and Role**

* Abstracts token encryption from Application.
* Used before saving sensitive tokens in DB.
* Ensures tokens only accessible through authorised service.

🔍 **Detailed Breakdown**  
Methods:

* EncryptAsync(string plaintext) → returns encrypted string.
* DecryptAsync(string ciphertext) → returns original plaintext.

⚠️ **Error Handling & Validation**

* Must handle invalid/expired ciphertext gracefully.
* Ensure deterministic encryption (same input, same output) is **not** used → must be non-deterministic with IV.

🔐 **Security Review**

* Must use strong AES-256 or equivalent.
* Keys must be stored in Key Vault, not code/config.
* Never log plaintext or ciphertext.

⚡ **Performance & Reliability**

* Encryption lightweight compared to password hashing.
* Reliability depends on secure key management.

📊 **Observability**

* Log encryption/decryption failures only (without data).

🧪 **Testability & Coverage**  
Suggested tests:

1. Encrypt + Decrypt returns same plaintext.
2. Decrypt with wrong key → fails.
3. Invalid ciphertext → handled safely.

🧹 **Code Smells**

* None at interface level.

🔧 **Refactoring Suggestions**

* Consider support for binary token formats.

📜 **Contracts & Compatibility**

* Central to OAuth token security.
* Breaking changes ripple across token storage and retrieval flows.

✅ **Confidence**  
High – clear, critical security abstraction.

**File Review**

**File:** Application/Interfaces/Services/IAccessService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~160

🧾 **ELI5**  
This defines the contract for checking whether a user has access to organisations and endpoints, and for assigning roles.

🎯 **Purpose and Role**

* Central access control service in the Application layer.
* Provides a unified way to:
  + Check permissions.
  + Assign roles.
  + Manage organisation-level access.
* Used by command/query handlers that enforce authorisation rules.

🔍 **Detailed Breakdown**  
Key methods:

* IsPlatformAdminAsync(userId) → checks if user is platform-wide admin.
* CanManageOrganisationAsync(userId, orgId) → checks if user can manage a given organisation.
* CanTriggerEndpointAsync(userId, orgId) → checks if user can run an endpoint ingestion.
* AssignRoleAsync(userId, orgId, roleId) → grants/replaces a role in an organisation.
* GetDefaultRoleIdAsync() → retrieves the default role for new users.
* IsAdmin(roleId) → checks if a role represents admin.

⚠️ **Error Handling & Validation**

* No explicit error handling in contract.
* Implementations must handle:
  + Unknown user/org/role gracefully.
  + Prevent duplicate role assignments.

🔐 **Security Review**

* Critical service: enforces RBAC (role-based access control).
* Must ensure role assignment requires elevated privileges.
* Incorrect implementation → privilege escalation risk.

⚡ **Performance & Reliability**

* Permission checks may be frequent.
* Should be cached where possible (role lookups).
* Must be strongly consistent for security-sensitive ops.

📊 **Observability**

* Implementations should log:
  + Role assignments.
  + Failed access attempts.

🧪 **Testability & Coverage**  
Suggested tests:

1. Platform admin check returns true only for admins.
2. Org admin can manage only their org.
3. AssignRole updates roles correctly.
4. Default role retrieved for onboarding.
5. IsAdmin returns correct result for admin vs non-admin roles.

🧹 **Code Smells**

* **Medium:** Role assignment and permission checks combined in one service (SRP overlap).
* **Low:** Boolean-based checks may lack granularity (e.g., "which permissions does user have?").

🔧 **Refactoring Suggestions**

* Split into:
  + IAuthorisationService (checks).
  + IRoleAssignmentService (mutations).
  + Effort: Medium, Priority: 3.
* Introduce GetUserPermissionsAsync for richer checks.
  + Effort: Medium, Priority: 2.

📜 **Contracts & Compatibility**

* Critical for all Application-layer handlers.
* Any change cascades to commands/queries across the app.

✅ **Confidence**  
High – clear, critical security contract.

**File Review**

**File:** Application/Interfaces/Services/ICallLogService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~230

🧾 **ELI5**  
This defines the contract for recording and retrieving API call logs and statistics for organisations and tenants.

🎯 **Purpose and Role**

* Central service for auditing external API interactions.
* Provides persistence and retrieval of:
  + Individual API call logs.
  + Aggregated statistics.
* Used by dashboard, ingestion monitoring, and error reporting.

🔍 **Detailed Breakdown**  
Key methods:

* **Write:**
  + LogCallAsync(callTime, orgId, userId?, endpoint, statusCode, isSuccess, rowsInserted = 0, errorMessage?) → records a call.
* **Read:**
  + GetLogsAsync(orgId) → retrieves logs for an organisation.
  + GetTenantLogsAsync(tenantId, userId) → logs for a tenant + user.
  + GetLatestStatsAsync(callGroupIds) → aggregated stats (rows inserted, success rate).

⚠️ **Error Handling & Validation**

* No explicit validation in contract.
* Implementation must:
  + Enforce valid HTTP status codes.
  + Sanitize errorMessage to avoid leaking secrets.

🔐 **Security Review**

* Call logs may contain sensitive error messages.
* Must sanitize to prevent token leakage.
* Should align with PII handling rules (mask email/user data).

⚡ **Performance & Reliability**

* High volume logging possible during ingestion.
* Requires batching or async insert.
* Queries should be paginated; indexes on OrgId, CallTime essential.

📊 **Observability**

* Service itself enables observability.
* Should log failures to write logs (rare but critical).

🧪 **Testability & Coverage**  
Suggested tests:

1. LogCall inserts valid entry.
2. Retrieve logs for org returns expected set.
3. Tenant log retrieval scoped to correct user.
4. Aggregated stats computed correctly.
5. Error message sanitisation enforced.

🧹 **Code Smells**

* **Medium:** errorMessage is free-form string → unstructured and risky.
* **Low:** No contract for pagination → logs could grow unbounded.

🔧 **Refactoring Suggestions**

* Replace free-form errorMessage with structured error codes + details.
  + Effort: Medium, Priority: 4.
* Add paging parameters to GetLogsAsync.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Contract underpins dashboard + monitoring.
* Changes ripple into Dashboard and Ingestion features.

✅ **Confidence**  
High – clear, aligns with ADR for observability.

**File Review**

**File:** Application/Interfaces/Services/IDashboardService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~90

🧾 **ELI5**  
This defines the contract for retrieving a user’s dashboard summary, which aggregates key information from across the system.

🎯 **Purpose and Role**

* Provides access to dashboard view data for a user.
* Used by Application handlers that populate the home/dashboard pages.
* Abstracts dashboard aggregation logic from UI.

🔍 **Detailed Breakdown**  
Methods:

* GetDashboardDataAsync(Guid userId) → retrieves a DashboardViewModel for the user.

⚠️ **Error Handling & Validation**

* Contract doesn’t define behaviour for invalid/missing users.
* Implementation must handle:
  + Non-existent user IDs.
  + Empty dashboard scenario.

🔐 **Security Review**

* Dashboard data may include sensitive org/ingestion stats.
* Must enforce user-level access control.

⚡ **Performance & Reliability**

* Dashboard summary may query multiple repos (logs, orgs, ingestion).
* Should be cached or aggregated efficiently.

📊 **Observability**

* Failures in dashboard build should be logged (with userId).

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid user returns populated DashboardViewModel.
2. User with no organisations → empty dashboard.
3. Invalid user ID → handled gracefully.

🧹 **Code Smells**

* **Low:** Contract is minimal; could be extended to support partial dashboard refresh.

🔧 **Refactoring Suggestions**

* Add overload for fetching dashboard by organisation.
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Defines API contract for dashboard.
* Changes ripple through Web UI and Home API.

✅ **Confidence**  
High – simple, well-scoped contract.

**File Review**

**File:** Application/Interfaces/Services/IDataIngestionService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This defines the contract for ingesting data from Xero (or other external sources) into RoadmApp.

🎯 **Purpose and Role**

* Provides abstraction for ingestion workflows.
* Allows Application layer to trigger ingestion jobs for tenants.
* Supports onboarding new organisations with initial data sync.

🔍 **Detailed Breakdown**  
Key methods:

* IngestEndpointsAsync(userId, tenantId) → triggers ingestion run for configured endpoints of a tenant.
* OnboardOrganisationsAsync(userId, tenantId) → performs initial ingestion for a new organisation.

⚠️ **Error Handling & Validation**

* Returns Result<T> wrappers for success/failure.
* No explicit contract for partial failures (e.g., one endpoint fails).
* Must enforce:
  + Valid tenant and user IDs.
  + Idempotency for onboarding (avoid duplicate ingestion).

🔐 **Security Review**

* Operates with OAuth tokens → must ensure they are retrieved securely via TokenRepository + Encryption.
* Logs must not contain token values.
* Access must be restricted (only authorised users trigger ingestion).

⚡ **Performance & Reliability**

* Ingestion is long-running and resource-intensive.
* Must support cancellation tokens for safe termination.
* Should be resilient to transient Xero API failures (retry with backoff).

📊 **Observability**

* Implementations should log per-endpoint ingestion status.
* Must record metrics (rows inserted, duration, errors).

🧪 **Testability & Coverage**  
Suggested tests:

1. Trigger ingestion for valid tenant → success result.
2. Invalid tenant → failure result.
3. Partial ingestion failure → logged and surfaced.
4. Onboard runs once only → idempotency check.
5. CancellationToken stops ingestion mid-run.

🧹 **Code Smells**

* **Medium:** Interface conflates two responsibilities (general ingestion + onboarding).

🔧 **Refactoring Suggestions**

* Split into:
  + IIngestionService (ongoing ingestion).
  + IOnboardingService (initial sync).
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Contract central to ingestion ADR.
* Breaking changes cascade through ingestion handlers and jobs.

✅ **Confidence**  
High – clear, but should separate onboarding from ingestion for SRP.

**File Review**

**File:** Application/Interfaces/Services/IEndpointConfigService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~160

🧾 **ELI5**  
This defines the contract for retrieving endpoint configurations that describe which Xero endpoints can be ingested and how they are set up.

🎯 **Purpose and Role**

* Abstracts endpoint configuration retrieval.
* Provides metadata for ingestion and UI display.
* Used by ingestion jobs, dashboard, and API explorers.

🔍 **Detailed Breakdown**  
Key methods:

* GetAllAsync() → retrieves all endpoint configs.
* GetInfoAsync() → retrieves endpoint info (for API/UI display).
* GetByNameAsync(name) → retrieves config by logical name.

⚠️ **Error Handling & Validation**

* Returns null when config not found.
* Implementation must enforce uniqueness of endpoint names.

🔐 **Security Review**

* Endpoint configs may contain sensitive scopes.
* Must not log full config details if they include sensitive data.

⚡ **Performance & Reliability**

* Endpoint configs are relatively static → should be cached.
* Index on endpoint name required.

📊 **Observability**

* Log if endpoint config is missing for a requested name.

🧪 **Testability & Coverage**  
Suggested tests:

1. Retrieve all → returns full list.
2. Retrieve by valid name → returns config.
3. Retrieve invalid name → returns null.
4. Duplicate name → fails in repo layer.

🧹 **Code Smells**

* **Low:** Contract is read-only; no methods for mutation → consistent but may limit flexibility.

🔧 **Refactoring Suggestions**

* Consider separation into IEndpointConfigReader and IEndpointConfigWriter if write operations required later.
  + Effort: Medium, Priority: 2.

📜 **Contracts & Compatibility**

* Supports ingestion and dashboard features.
* Changes ripple into Xero ingestion handlers.

✅ **Confidence**  
High – well-scoped, consistent with endpoint config ADR.

**File Review**

**File:** Application/Interfaces/Services/IOrganisationService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~300

🧾 **ELI5**  
This defines the contract for managing organisations (tenants) – retrieving details, editing them, assigning users, and listing organisations available to a user.

🎯 **Purpose and Role**

* Provides organisation-level operations in the Application layer.
* Used by command/query handlers for organisation CRUD and user assignment.
* Bridges Application and persistence for tenant management.

🔍 **Detailed Breakdown**  
Key methods:

* GetInfoAsync(orgId) → retrieves OrganisationInfoDto for a single organisation.
* EditOrganisationAsync(orgId, orgName) → edits org name.
* AssignUsersAsync(orgId, List<Guid> userIds) → assigns users to org.
* GetUserOrganisationsAsync(userId) → retrieves all organisations accessible by a user.

⚠️ **Error Handling & Validation**

* Returns Result<T> wrappers for consistency.
* Must handle:
  + Non-existent orgs.
  + Duplicate user assignments.
  + Race conditions during edits.

🔐 **Security Review**

* User assignment is security-sensitive → must require elevated privileges.
* GetUserOrganisationsAsync must enforce caller identity (no cross-tenant leakage).

⚡ **Performance & Reliability**

* Queries must be indexed on OrgId and UserId.
* Bulk user assignment should be batched.
* Edits should be transactional.

📊 **Observability**

* Implementations should log:
  + Org edits.
  + User assignments.
  + Failed retrievals.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid org retrieval → returns info.
2. Edit org → name updated.
3. Assign multiple users → success.
4. Duplicate assignment → no duplicates created.
5. User with no orgs → empty list.

🧹 **Code Smells**

* **Medium:** Interface mixes CRUD operations and user assignment.
* **Low:** Only supports renaming org, not deletion.

🔧 **Refactoring Suggestions**

* Split into IOrganisationService (CRUD) and IUserOrganisationService (assignments).
  + Effort: Medium, Priority: 3.
* Add DeleteOrganisationAsync.
  + Effort: Small, Priority: 4.

📜 **Contracts & Compatibility**

* Contract central to multi-tenant logic.
* Breaking changes ripple through auth, ingestion, and dashboard features.

✅ **Confidence**  
High – clear, critical to multi-tenancy.

**File Review**

**File:** Application/Interfaces/Services/IPollingService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This defines the contract for running scheduled polling jobs that fetch new data from Xero and store it in RoadmApp.

🎯 **Purpose and Role**

* Provides abstraction for executing polling tasks.
* Central point for scheduled/background ingestion.
* Used by job scheduler to trigger tenant-level polling.

🔍 **Detailed Breakdown**  
Key method:

* PollAsync(userId, cancellationToken) → triggers a polling run for a user’s tenants, retrieving new data and persisting it.

⚠️ **Error Handling & Validation**

* Returns Task only → no direct success/failure feedback.
* Implementation must handle:
  + Invalid userId.
  + Token expiry mid-poll.
  + Partial ingestion failures.

🔐 **Security Review**

* Operates with OAuth tokens → must fetch via TokenRepository and decrypt securely.
* Polling must only run for authorised tenants.

⚡ **Performance & Reliability**

* Long-running, high-throughput task.
* Must support:
  + CancellationToken.
  + Retry/backoff for Xero API failures.
  + Batching to prevent overload.

📊 **Observability**

* Must log:
  + Poll start/end.
  + Per-endpoint results.
  + Failures with safe sanitisation.
* Metrics should track rows ingested, duration, failures.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid poll inserts data.
2. Expired token → refresh attempted.
3. Partial endpoint failure logged but continues others.
4. Cancellation stops job mid-run.
5. High-volume polling scales correctly.

🧹 **Code Smells**

* **Medium:** Single method contract is too coarse-grained.

🔧 **Refactoring Suggestions**

* Expand to include:
  + PollTenantAsync(tenantId)
  + PollEndpointAsync(endpointKey)
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Critical for ingestion ADR.
* Any breaking changes impact job scheduler and background workers.

✅ **Confidence**  
High – simple but central to system reliability.

**File Review**

**File:** Application/Interfaces/Services/IPollingSettingsService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~300

🧾 **ELI5**  
This defines the contract for managing polling schedules for organisations – deciding how often background ingestion jobs should run.

🎯 **Purpose and Role**

* Provides abstraction for retrieving and updating polling settings.
* Bridges Application layer with repository logic.
* Used by job scheduler and dashboard.

🔍 **Detailed Breakdown**  
Key methods:

* GetAsync(orgId) → polling settings for one org.
* GetManyAsync(orgIds) → polling settings for multiple orgs.
* GetAllAsync() → polling settings for all orgs.
* UpsertAsync(orgId, frequency, runTime?) → create/update schedule.

⚠️ **Error Handling & Validation**

* Uses Result<T> wrapper for consistency.
* No contract for invalid combinations (e.g., Hourly + runTime).
* Must enforce rules:
  + Hourly → runTime null.
  + Daily → runTime required.

🔐 **Security Review**

* Settings not sensitive, but malicious inputs could cause denial of service (over-frequent polling).
* Must validate and restrict allowed frequencies.

⚡ **Performance & Reliability**

* Queries are lightweight.
* Bulk retrieval should be cached for scheduler startup.
* Indexes on OrganisationId required.

📊 **Observability**

* Implementations should log:
  + Updates to schedules.
  + Invalid attempts.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid upsert persists schedule.
2. Hourly + runTime provided → rejected.
3. Daily + runTime missing → rejected.
4. Bulk retrieval returns correct mapping.
5. GetAllAsync returns empty when no orgs.

🧹 **Code Smells**

* **Medium:** Interface mixes retrieval and mutation responsibilities.
* **Low:** No explicit delete contract.

🔧 **Refactoring Suggestions**

* Add DeleteAsync(orgId).
  + Effort: Small, Priority: 3.
* Split into IPollingSettingsReader and IPollingSettingsWriter.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Core to background job scheduler.
* Breaking changes ripple across dashboard and scheduler services.

✅ **Confidence**  
High – well-scoped but could benefit from split responsibilities and stricter validation.

**File Review**

**File:** Application/Interfaces/Services/IPythonEtlBridge.cs  
**Layer/Type:** Application – Service Interface (ETL Bridge)  
**Status:** Reviewed  
**Tokens (approx.):** ~180

🧾 **ELI5**  
This defines the contract for running ETL jobs implemented in Python from within the .NET application. It acts as a bridge to external Python-based ingestion logic.

🎯 **Purpose and Role**

* Allows Application services to invoke Python ETL pipelines.
* Supports multiple named ETL profiles (e.g., RawToLanding, LandingToOds).
* Bridges RoadmApp with Python ingestion runtime.

🔍 **Detailed Breakdown**  
Key methods:

* RunAsync(profileName, tenantId, userId, batchId, sourceNames) → runs an ETL profile by name.
* RunRawToLandingAsync(...) → specialised call for Raw → Landing load.
* RunLandingToOdsAsync(...) → specialised call for Landing → ODS load.

⚠️ **Error Handling & Validation**

* Returns Task<string?> → success/failure indicated by result string.
* No explicit contract for error codes.
* Implementation must:
  + Validate profile name.
  + Handle Python process failures robustly.

🔐 **Security Review**

* ETL jobs may access sensitive financial/PII data.
* Must ensure sandboxing of Python processes.
* Logs must not expose raw payloads or credentials.

⚡ **Performance & Reliability**

* ETL jobs can be long-running.
* Must support cancellation tokens.
* Batch ID required for correlation and retry.

📊 **Observability**

* Should log ETL invocation, duration, and outcome.
* Must track per-profile success/failure metrics.

🧪 **Testability & Coverage**  
Suggested tests:

1. Run valid profile returns non-null result.
2. Invalid profile → failure.
3. RawToLanding executes correct Python script.
4. LandingToOds executes correct Python script.
5. Cancellation token halts ETL job.

🧹 **Code Smells**

* **Medium:** Returns raw string result → lacks structured response (status, error, metadata).
* **Low:** Methods mix generic and specific ETL calls.

🔧 **Refactoring Suggestions**

* Replace string? with structured result DTO.
  + Effort: Medium, Priority: 4.
* Split into profile-based generic bridge and separate typed services.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Central to ingestion ADR.
* Breaking changes ripple through ingestion jobs and pipelines.

✅ **Confidence**  
High – clear but needs stronger typing and structured responses.

**File Review**

**File:** Application/Interfaces/Services/IUserService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~600 (large, many operations)

🧾 **ELI5**  
This defines the contract for everything related to users – login, registration, password management, role assignments, org access, invitations, and profile edits.

🎯 **Purpose and Role**

* Primary interface for user management in the Application layer.
* Covers:
  + Authentication (login).
  + Registration + admin creation.
  + Password reset/change flows.
  + Org/role assignments.
  + Profile management.
* Used by almost all user-related handlers.

🔍 **Detailed Breakdown**  
Key categories of methods:

1. **Authentication**
   * AuthenticateAsync(username, password) → validates credentials.
2. **Registration**
   * RegisterAsync(UserRegistrationRequest) → new user.
   * CreateUserAsync(UserCreationRequest) → admin creates user.
   * InviteUserAsync(UserCreationRequest) → send invitation email.
   * InitAdminAsync(UserCreationRequest) → bootstrap admin.
3. **Password Management**
   * ChangePasswordAsync(userId, newPassword) → change logged-in user’s password.
   * SendPasswordResetAsync(email) → send reset email.
   * GeneratePasswordResetTokenAsync(userId, lifetime) → create reset token.
   * ResetPasswordAsync(PasswordResetRequest) → reset via token.
4. **Organisation + Role Management**
   * GetUserRolesAsync(userId) → mapping of user’s roles.
   * AssignRoleAsync(userId, orgId, roleId) → grant/change role.
   * RemoveUserOrgAccessAsync(userId, orgId) → revoke access.
   * GetUserOrgNamesAsync(userId) → user’s orgs.
5. **Profile Management**
   * EditProfileAsync(userId, fullName, email) → edit profile.

⚠️ **Error Handling & Validation**

* Uses Result<T> wrapper throughout.
* Must enforce:
  + Strong password rules.
  + Email uniqueness.
  + Admin-only access to admin functions.

🔐 **Security Review**

* Handles highly sensitive data:
  + Passwords must be hashed securely (BCrypt/Argon2).
  + Reset tokens must be random, one-time use, stored hashed.
  + Role assignment must be restricted to admins.
* Must avoid logging passwords or tokens.

⚡ **Performance & Reliability**

* Many operations involve DB + email I/O.
* Should be async, resilient to email failures.
* Indexes required on Username, Email, UserId.

📊 **Observability**

* Implementations should log:
  + Failed login attempts.
  + Password resets.
  + Role/Org changes.

🧪 **Testability & Coverage**  
Suggested tests:

1. Login with valid credentials → success.
2. Invalid password → failure.
3. Register new user → stored with hashed password.
4. Duplicate email → rejected.
5. Password reset token flow works end-to-end.
6. AssignRole only allowed by admins.
7. RemoveUserOrgAccess revokes properly.
8. EditProfile updates fields.

🧹 **Code Smells**

* **High:** Interface is huge (God interface). Mixes authentication, password mgmt, org mgmt, profile mgmt.
* **Medium:** Overlaps with AccessService and OrganisationService.
* **Low:** Some methods return only bool → could use structured responses.

🔧 **Refactoring Suggestions**

* Split into:
  + IAuthService (auth/login/passwords).
  + IUserManagementService (registration, profile).
  + IUserOrgService (org/role assignments).
  + Effort: Large, Priority: 3.
* Replace plain bool results with structured DTOs.
  + Effort: Medium, Priority: 2.

📜 **Contracts & Compatibility**

* Central contract for user lifecycle.
* Breaking changes ripple across all user-related features.

✅ **Confidence**  
High – comprehensive, but too broad.

**File Review**

**File:** Application/Interfaces/Services/IXeroEntitySyncService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~70

🧾 **ELI5**  
This defines the contract for synchronising a single Xero entity (like an invoice) into RoadmApp.

🎯 **Purpose and Role**

* Provides a fine-grained sync method for Xero entities.
* Allows Application layer to fetch and persist one record without a full ingestion run.
* Useful for targeted updates, retries, or webhooks.

🔍 **Detailed Breakdown**  
Method:

* SyncInvoiceAsync(tenantId, invoiceId) → fetches invoice by ID and stores it in raw table for the tenant.

⚠️ **Error Handling & Validation**

* Returns Task.
* No structured error reporting → must rely on exceptions or logs.
* Must validate tenant and invoice IDs.

🔐 **Security Review**

* Operates with OAuth tokens → must retrieve securely and avoid logging.
* Invoice data may contain PII/financial info → must protect at rest.

⚡ **Performance & Reliability**

* Lightweight compared to full ingestion.
* Still requires resilience to transient Xero API failures (retry).

📊 **Observability**

* Should log:
  + Invoice ID synced.
  + Failures with safe sanitisation.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid invoice ID → syncs successfully.
2. Invalid ID → fails gracefully.
3. API timeout → retries.
4. Tenant without access → rejected.

🧹 **Code Smells**

* **Low:** Single-method interface → narrow contract.

🔧 **Refactoring Suggestions**

* Consider expanding to other entities (Contacts, Payments).
  + Effort: Medium, Priority: 3.
* Return structured Result<T> instead of bare Task.
  + Effort: Small, Priority: 4.

📜 **Contracts & Compatibility**

* Contract part of webhook-triggered ingestion ADR.
* Expanding scope may affect existing consumers.

✅ **Confidence**  
High – narrow but clear purpose.

**File Review**

**File:** Application/Interfaces/Services/IXeroEtlService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~80

🧾 **ELI5**  
This defines the contract for running ETL (Extract, Transform, Load) jobs from Xero raw data into RoadmApp’s ODS (Operational Data Store).

🎯 **Purpose and Role**

* Provides orchestration for ETL workflows.
* Supports tenant-level and global ETL runs.
* Used by background jobs and admin commands to refresh data.

🔍 **Detailed Breakdown**  
Methods:

* RunEtlAsync(endpointConfig, tenantId, userId) → runs ETL for specific endpoint + tenant.
* RunGlobalEtlAsync(tenantId, userId) → runs ETL across all configured endpoints for a tenant.

⚠️ **Error Handling & Validation**

* Returns Task only → no structured result.
* Implementation must catch failures and report logs.
* Must validate endpoint config and tenantId.

🔐 **Security Review**

* Operates with OAuth tokens → must decrypt securely.
* ETL results may contain sensitive financial/PII data → must be secured at rest.

⚡ **Performance & Reliability**

* ETL is long-running and resource-heavy.
* Must support retries, backoff, and cancellation.

📊 **Observability**

* Must log ETL start/end, per-endpoint outcomes.
* Metrics needed (duration, rows transformed).

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid ETL run completes.
2. Invalid endpoint config → fails gracefully.
3. Tenant without access → rejected.
4. CancellationToken halts ETL mid-run.

🧹 **Code Smells**

* **Low:** Contract only returns Task → no structured result.

🔧 **Refactoring Suggestions**

* Replace bare Task with Task<Result<EtlReport>>.
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Part of ingestion/ETL ADR.
* Changes ripple into job scheduler and background jobs.

✅ **Confidence**  
High – small, clear, but needs structured results.

**File Review**

**File:** Application/Interfaces/Services/IXeroScopeService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~70

🧾 **ELI5**  
This defines the contract for retrieving all active OAuth scopes configured for Xero integration.

🎯 **Purpose and Role**

* Provides Application layer access to scope configuration.
* Used during login/authorisation to ensure correct scopes requested.
* Bridges Application with repository of XeroScope entities.

🔍 **Detailed Breakdown**  
Method:

* GetActiveScopesAsync() → returns list of active XeroScope records.

⚠️ **Error Handling & Validation**

* Contract assumes repository enforces active/inactive status.
* Implementation must handle empty results.

🔐 **Security Review**

* Scopes are not secrets but incorrect configuration may grant excessive permissions.
* Must align with ADR requiring scopes as collections, not free-form strings.

⚡ **Performance & Reliability**

* Read-only, lightweight operation.
* Should be cached for performance.

📊 **Observability**

* Should log if no active scopes are found (misconfiguration).

🧪 **Testability & Coverage**  
Suggested tests:

1. Active scopes exist → list returned.
2. No active scopes → empty list.
3. Misconfigured scopes → handled gracefully.

🧹 **Code Smells**

* **Low:** Very narrow contract; only supports active scope retrieval.

🔧 **Refactoring Suggestions**

* Add GetAllScopesAsync() and GetByIdAsync() for completeness.
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* Central to auth flow.
* Any change ripples into OAuth login handler.

✅ **Confidence**  
High – simple, consistent, but could be expanded slightly.

**File Review**

**File:** Application/Interfaces/Services/IXeroTokenService.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~600 (large, multi-purpose contract)

🧾 **ELI5**  
This defines the contract for managing all aspects of Xero OAuth2 tokens: retrieving, refreshing, storing, revoking, and building login flows.

🎯 **Purpose and Role**

* Central service for token lifecycle management.
* Handles:
  + Access + refresh token retrieval.
  + Token refresh + persistence.
  + Token revocation/disconnection.
  + Building login/redirect URIs.
* Used by auth, ingestion, and webhook services.

🔍 **Detailed Breakdown**  
Key methods:

1. **Token Retrieval**

* GetValidTokenAsync(userId) → returns non-expired token, refresh if needed.
* RetrieveTokenAsync(userId) → fetch most recent token for user.

1. **Token Storage**

* StoreTokenAsync(token, userId) → persist access/refresh tokens.
* DestroyTokenForUserAsync(userId) → remove all tokens.
* RemoveTenantAsync(userId, tenantId) → revoke tenant-level tokens.

1. **User ↔ Token Mapping**

* GetAllUsersWithTokensAsync() → returns user IDs with tokens.
* GetUserIdForTenantAsync(tenantId) → resolve which user owns tenant’s token.

1. **Auth Flow**

* BuildLoginUri(state, scopes, redirectUri) → constructs OAuth2 login URI.
* AuthoriseAsync(code, userId) → exchanges code for token + stores it.

1. **Tenant Management**

* DisconnectTenantAsync(userId, tenantId) → disconnects tenant + revokes token.
* IsTenantConnectedAsync(userId, orgId) → check connection.

⚠️ **Error Handling & Validation**

* Uses Result<T> for structured failures.
* Implementation must handle:
  + Token expiry.
  + Invalid tenant → safe failure.
  + Multiple tokens per user/tenant.

🔐 **Security Review**

* Handles highly sensitive OAuth tokens:
  + Must be encrypted at rest.
  + Must never log tokens.
* Login URI construction must validate state + redirectUri (prevent open redirect).
* Revocation must ensure tenant isolation.

⚡ **Performance & Reliability**

* Token refresh is frequent and latency-sensitive.
* Must implement retry with backoff for Xero API.
* Index on UserId + TenantId required.

📊 **Observability**

* Must log:
  + Token refresh attempts.
  + Revocations/disconnects.
  + Failed login URIs.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid token retrieved.
2. Expired token → refresh flow succeeds.
3. Invalid refresh → failure result.
4. Disconnect tenant removes token.
5. BuildLoginUri produces correct string.
6. AuthoriseAsync stores token.

🧹 **Code Smells**

* **High:** Very large interface → mixes auth flow, storage, tenant management.
* **Medium:** String parameters for scopes/state → risk of errors.
* **Low:** Overlaps with TokenRepository (duplication of storage logic).

🔧 **Refactoring Suggestions**

* Split into:
  + IXeroAuthService (login/authorisation).
  + IXeroTokenStore (storage).
  + IXeroTenantService (tenant management).
  + Effort: Large, Priority: 3.
* Replace string-based scopes with typed collection.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Critical for OAuth integration ADR.
* Breaking changes ripple into all Xero-related features.

✅ **Confidence**  
High – well-aligned but too broad.

**File Review**

**File:** Application/Interfaces/Services/IXeroWebhookQueue.cs  
**Layer/Type:** Application – Service Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~120

🧾 **ELI5**  
This defines the contract for queueing Xero webhook events for later background processing.

🎯 **Purpose and Role**

* Decouples webhook reception from processing.
* Provides methods to:
  + Enqueue raw webhook payloads.
  + Dequeue and stream webhook payloads for processing.
* Supports async background workers.

🔍 **Detailed Breakdown**  
Methods:

* EnqueueAsync(eventPayload) → push webhook payload to queue.
* DequeueAllAsync() → retrieve all queued payloads as async stream.

⚠️ **Error Handling & Validation**

* No explicit contract for invalid payloads.
* Must enforce payload schema before enqueue.
* Implementation must handle empty queue safely.

🔐 **Security Review**

* Webhook payloads may contain sensitive financial data.
* Must validate webhook signatures before enqueue.
* Logs must not expose raw payloads.

⚡ **Performance & Reliability**

* Queue must scale to handle burst traffic from Xero.
* Should support distributed workers (e.g., Redis, Azure Queue).
* Must ensure at-least-once delivery (avoid event loss).

📊 **Observability**

* Log enqueue/dequeue operations (with metadata, not payload).
* Monitor queue size and processing latency.

🧪 **Testability & Coverage**  
Suggested tests:

1. Enqueue valid payload → retrievable via DequeueAll.
2. Empty queue → returns empty stream.
3. Invalid payload → rejected.
4. High-volume enqueue/dequeue → stable.

🧹 **Code Smells**

* **Low:** Very minimal contract, assumes single queue type.

🔧 **Refactoring Suggestions**

* Add support for queue priorities or partitioning (per tenant).
  + Effort: Medium, Priority: 3.
* Return structured result instead of raw string payloads.
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Part of webhook processing ADR.
* Changes ripple into webhook handlers and background processors.

✅ **Confidence**  
High – simple but critical to reliable webhook handling.

**File Review**

**File:** Application/Interfaces/Utilities/IWebUrlProvider.cs  
**Layer/Type:** Application – Utility Interface  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This defines the contract for building fully qualified URLs within the web app (e.g., callback links for emails, OAuth redirects).

🎯 **Purpose and Role**

* Provides a utility to generate consistent URLs for use across the app.
* Keeps Application layer decoupled from Infrastructure/web config.
* Supports building environment-specific links (dev/test/prod).

🔍 **Detailed Breakdown**  
Method:

* GetBaseUrl() → returns app’s root URL.
* BuildUrl(path) → concatenates base URL with given relative path.

⚠️ **Error Handling & Validation**

* Contract assumes valid base URL.
* Must validate path format (avoid double slashes, missing leading slash).

🔐 **Security Review**

* Critical for OAuth/email flows → redirect URIs must be validated.
* Must prevent open redirect vulnerabilities.
* Should enforce HTTPS in production.

⚡ **Performance & Reliability**

* Lightweight operation.
* Relies on correct config injection.

📊 **Observability**

* Misconfigurations (e.g., wrong base URL) should be logged.

🧪 **Testability & Coverage**  
Suggested tests:

1. BuildUrl("/login") → returns correct full URL.
2. Ensure HTTPS enforced in production.
3. Invalid/missing base URL → throws or logs error.

🧹 **Code Smells**

* **Low:** Narrow contract – only supports simple concatenation.

🔧 **Refactoring Suggestions**

* Extend to support building signed/expiring URLs.
  + Effort: Medium, Priority: 2.
* Add environment awareness (different base URLs per tenant).
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Supports cross-cutting concerns (auth, email).
* Breaking changes affect all email templates and redirects.

✅ **Confidence**  
High – small, clear, critical for URL building consistency.

**File Review**

**File:** Application/Services/Access/AccessService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~400

🧾 **ELI5**  
This class implements the access control logic, deciding whether a user can manage an organisation, trigger ingestion, or has admin rights.

🎯 **Purpose and Role**

* Implements IAccessService.
* Centralises role-based access control (RBAC).
* Provides checks for:
  + Platform admin status.
  + Organisation management rights.
  + Endpoint trigger rights.
  + Default role resolution.
* Used across commands/queries to enforce security boundaries.

🔍 **Detailed Breakdown**  
Dependencies:

* IPermissionRepository → checks role/permission assignments.
* IRoleRepository → retrieves default role.
* IOrganisationRepository → ensures user-org-role links.
* ILogger<AccessService> → logs operations.

Key methods:

* IsPlatformAdminAsync(userId) → checks platform admin status.
* CanManageOrganisationAsync(userId, orgId) → checks organisation admin role.
* CanTriggerEndpointAsync(userId, orgId) → checks endpoint trigger permissions.
* AssignRoleAsync(userId, orgId, roleId) → assigns user to org + role.
* GetDefaultRoleIdAsync() → returns default role GUID.
* IsAdmin(roleId) → checks if role represents admin.

⚠️ **Error Handling & Validation**

* Uses async Result<T> patterns consistently.
* Logs failures (e.g., missing role).
* Must ensure:
  + Null role IDs handled.
  + Duplicate assignments prevented.

🔐 **Security Review**

* Critical service enforcing RBAC.
* Relies on repositories for data integrity.
* Risks:
  + Misconfigured default role could grant unintended privileges.
  + AssignRole must be restricted to privileged users.

⚡ **Performance & Reliability**

* Frequent permission checks → must be efficient.
* Should cache role lookups.
* DB indexes on UserId + OrgId required.

📊 **Observability**

* Logs role assignment, missing roles, and failures.
* Could benefit from structured log fields (UserId, OrgId).

🧪 **Testability & Coverage**  
Suggested tests:

1. User with admin role → IsPlatformAdmin = true.
2. Non-admin → false.
3. CanManageOrganisation returns true for org admin, false otherwise.
4. CanTriggerEndpoint enforces role scope.
5. AssignRole adds mapping.
6. GetDefaultRoleId returns expected GUID.

🧹 **Code Smells**

* **Medium:** Service handles both checks and assignments (SRP overlap).
* **Low:** Logging may use string interpolation instead of structured placeholders.

🔧 **Refactoring Suggestions**

* Split into:
  + IAuthorisationService (checks).
  + IRoleAssignmentService (mutations).
  + Effort: Medium, Priority: 3.
* Standardise structured logging (logger.LogInformation("User {UserId} ...")).
  + Effort: Small, Priority: 2.

📜 **Contracts & Compatibility**

* Implements IAccessService.
* Changes ripple into every command/query enforcing permissions.

✅ **Confidence**  
High – robust implementation of access control, but mixes responsibilities.

**File Review**

**File:** Application/Services/Auth/UserService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~2500 (large, multi-responsibility class)

🧾 **ELI5**  
This class manages everything about users: login, registration, password resets, profile updates, and organisation/role assignments.

🎯 **Purpose and Role**

* Implements IUserService.
* Primary orchestration service for user lifecycle.
* Provides a single entry point for:
  + Authentication.
  + Registration (self-service + admin).
  + Password management.
  + Invitations + onboarding.
  + Role and organisation assignments.
  + Profile editing.

🔍 **Detailed Breakdown**  
Dependencies (injected):

* IUserRepository → persistence of user records.
* IPasswordHasher → secure password hashing/verification.
* IPasswordResetTokenRepository → reset token lifecycle.
* IEmailService → send invitations and reset emails.
* IAccessService → role + permission checks.
* ILogger<UserService> → logging.
* Others: IUnitOfWorkFactory, IOrganisationRepository, etc.

Key methods (grouped):

1. **Authentication**
   * AuthenticateAsync(username, password) → validates login, returns Result<RoadmappUser>.
2. **Registration & Creation**
   * RegisterAsync(request) → new user self-service.
   * CreateUserAsync(request) → admin creation.
   * InviteUserAsync(request) → sends invitation with reset link.
   * InitAdminAsync(request) → bootstrap admin user.
3. **Password Management**
   * ChangePasswordAsync(userId, newPassword) → updates stored hash.
   * SendPasswordResetAsync(email) → generates + emails reset token.
   * GeneratePasswordResetTokenAsync(userId, lifetime) → token creation.
   * ResetPasswordAsync(request) → verifies + updates password.
4. **Organisation & Role Management**
   * GetUserRolesAsync(userId) → retrieves roles per org.
   * AssignRoleAsync(userId, orgId, roleId) → sets role.
   * RemoveUserOrgAccessAsync(userId, orgId) → revokes access.
5. **Profile Management**
   * EditProfileAsync(userId, name, email) → updates profile.

⚠️ **Error Handling & Validation**

* Uses Result<T> for structured failures.
* Guards against null/invalid inputs.
* Validates uniqueness of usernames/emails.
* Risk: some methods rely on repository exceptions rather than explicit checks.

🔐 **Security Review**

* ✅ Passwords hashed before storage.
* ✅ Reset tokens managed via repository.
* ❌ Risk: tokens must be stored hashed, not plaintext.
* ✅ Email-based reset flow supported.
* ✅ Role assignment enforced via IAccessService.
* Risk: Logging must avoid sensitive values (passwords, tokens).

⚡ **Performance & Reliability**

* Most operations async, DB + email I/O.
* User + email lookups require indexes.
* High reliability needed for reset and onboarding flows.

📊 **Observability**

* Logs major events: registration, failed login, role changes.
* Could benefit from structured logging with UserId, OrgId.

🧪 **Testability & Coverage**  
Suggested tests:

1. Login with valid credentials.
2. Invalid password → fails.
3. Register user with duplicate email → fails.
4. Password reset flow end-to-end.
5. Role assignment restricted to admins.
6. Invite email contains correct URL.
7. Remove user org access revokes properly.

🧹 **Code Smells**

* **High:** God service – mixes auth, registration, password, role, and org mgmt.
* **Medium:** Overlaps with AccessService and OrganisationService.
* **Low:** Some methods return only bool instead of structured result.

🔧 **Refactoring Suggestions**

* Split into focused services:
  + IAuthService (login + password mgmt).
  + IUserManagementService (registration, profile).
  + IUserOrgService (org/role mgmt).
  + Effort: Large, Priority: 3.
* Standardise structured logging across methods.
  + Effort: Medium, Priority: 2.

📜 **Contracts & Compatibility**

* Implements IUserService.
* Breaking changes ripple across login, registration, and user handlers.

✅ **Confidence**  
High – class is comprehensive but too large. Needs decomposition for long-term maintainability.

**File Review**

**File:** Application/Services/CallLogging/CallLogService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~350

🧾 **ELI5**  
This class records API calls (like ingestion calls to Xero), and provides ways to query logs and statistics for organisations and tenants.

🎯 **Purpose and Role**

* Implements ICallLogService.
* Provides methods to:
  + Log API calls with outcome, status, and row count.
  + Retrieve logs for an organisation or tenant.
  + Aggregate stats (latest calls per group).
* Supports auditing, monitoring, and dashboard features.

🔍 **Detailed Breakdown**  
Dependencies:

* ICallLogRepository → persistence.
* ILogger<CallLogService> → logging.
* IUnitOfWorkFactory → transactional safety.

Key methods:

* LogCallAsync(callTime, orgId, userId?, endpoint, statusCode, isSuccess, rowsInserted, errorMessage?) → inserts a call log.
* GetLogsAsync(orgId) → retrieves logs for org.
* GetTenantLogsAsync(tenantId, userId) → logs for specific tenant + user.
* GetLatestStatsAsync(callGroupIds) → aggregated stats.

⚠️ **Error Handling & Validation**

* Wraps repository calls with Result<T>.
* Validates org/tenant existence indirectly via repository.
* Risks:
  + errorMessage free-form string could leak sensitive info.
  + No explicit paging contract → large log queries may overload memory.

🔐 **Security Review**

* Logs may contain sensitive endpoint names and error messages.
* Must sanitize errors to avoid leaking tokens/PII.
* Logs should be restricted by user permissions (no cross-tenant access).

⚡ **Performance & Reliability**

* Logging is frequent during ingestion.
* Must be optimised with async inserts and DB indexes.
* GetLogsAsync without paging risks performance issues.

📊 **Observability**

* Service itself contributes observability.
* Should log failures to persist logs.

🧪 **Testability & Coverage**  
Suggested tests:

1. LogCall inserts success record.
2. LogCall inserts failure record with error.
3. GetLogs returns only org-specific entries.
4. GetTenantLogs scoped correctly.
5. GetLatestStats aggregates rows correctly.

🧹 **Code Smells**

* **Medium:** Free-form errorMessage.
* **Medium:** No paging support → unbounded reads.
* **Low:** Service overlaps strongly with repository logic (thin wrapper).

🔧 **Refactoring Suggestions**

* Add pagination to log retrieval methods.
  + Effort: Medium, Priority: 4.
* Replace errorMessage with structured error model.
  + Effort: Medium, Priority: 3.
* Consider adding async bulk insert for high-frequency logging.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Implements ICallLogService.
* Contract supports dashboard features → changes ripple into monitoring UI.

✅ **Confidence**  
High – service is consistent, but log retrieval scalability and error sanitisation need attention.

**File Review**

**File:** Application/Services/Config/EndpointConfigService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~150

🧾 **ELI5**  
This class manages endpoint configuration records — defining which Xero endpoints are available and how they are set up.

🎯 **Purpose and Role**

* Implements IEndpointConfigService.
* Provides methods to retrieve and manage endpoint configurations.
* Used by ingestion services, dashboards, and API explorers.

🔍 **Detailed Breakdown**  
Dependencies:

* IEndpointConfigRepository → persistence of configs.
* ILogger<EndpointConfigService> → logging.

Key methods:

* GetAllAsync() → retrieves all endpoint configs.
* GetInfoAsync() → retrieves info (for UI/API display).
* GetByNameAsync(name) → retrieves specific config by name.

⚠️ **Error Handling & Validation**

* Returns Result<T> wrappers.
* Must enforce uniqueness of endpoint names.
* Logs errors when config missing.

🔐 **Security Review**

* Configs may include scope info → must not log sensitive data.
* Input name lookups should be sanitised to prevent injection.

⚡ **Performance & Reliability**

* Endpoint configs are relatively static.
* Should be cached for performance.

📊 **Observability**

* Logs when configs are missing or invalid.

🧪 **Testability & Coverage**  
Suggested tests:

1. GetAll returns all configs.
2. GetByName with valid name returns config.
3. GetByName with invalid name returns null/failure.
4. Repository error → handled gracefully.

🧹 **Code Smells**

* **Low:** Service is a thin wrapper around repository.

🔧 **Refactoring Suggestions**

* Introduce caching layer to reduce DB calls.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Implements IEndpointConfigService.
* Contract affects ingestion and dashboards.

✅ **Confidence**  
High – simple, well-scoped service.

**File Review**

**File:** Application/Services/Dashboard/DashboardService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~100

🧾 **ELI5**  
This class builds the dashboard summary for a user by querying logs and stats.

🎯 **Purpose and Role**

* Implements IDashboardService.
* Provides the main method for constructing a user’s DashboardViewModel.
* Consumed by dashboard query handlers.

🔍 **Detailed Breakdown**  
Dependencies:

* ICallLogRepository → retrieves stats and logs.
* ILogger<DashboardService> → logs failures.

Key method:

* GetDashboardDataAsync(userId) → builds and returns DashboardViewModel.

⚠️ **Error Handling & Validation**

* Wraps repository calls in Result<T>.
* Logs errors when stats cannot be retrieved.
* Does not validate userId directly (relies on repository).

🔐 **Security Review**

* Returns dashboard summary only for the given user.
* Must enforce caller identity match (to prevent cross-user leakage).

⚡ **Performance & Reliability**

* Dependent on log repository efficiency.
* Should be cached if dashboard queries are frequent.

📊 **Observability**

* Logs failures to build dashboard.
* Could benefit from structured logging (UserId).

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid user with logs → returns populated dashboard.
2. Valid user with no logs → returns empty dashboard.
3. Repository failure → returns failure result.

🧹 **Code Smells**

* **Low:** Thin wrapper over repository.

🔧 **Refactoring Suggestions**

* Introduce caching for performance.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Implements IDashboardService.
* Changes ripple into Dashboard query handlers.

✅ **Confidence**  
High – simple, well-aligned service.

**File Review**

**File:** Application/Services/DataIngestion/DataIngestionService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~700 (large service)

🧾 **ELI5**  
This class runs data ingestion from Xero, fetching API data for tenants, storing it as raw payloads, and producing ingestion reports.

🎯 **Purpose and Role**

* Implements IDataIngestionService.
* Coordinates end-to-end ingestion pipeline:
  + Calls Xero API for endpoint data.
  + Persists raw payloads.
  + Runs transformations.
  + Returns IngestionResponse.
* Used for both **scheduled polling** and **manual ingestion triggers**.

🔍 **Detailed Breakdown**  
Dependencies:

* IXeroApiClient → fetches data from Xero endpoints.
* IRawXeroPayloadRepository → persists raw payload JSON.
* IPollingSettingsRepository → retrieves polling configs.
* ICallLogService → records API call results.
* ILogger<DataIngestionService> → logging.
* IUnitOfWorkFactory → transactional safety.

Key methods:

* IngestEndpointsAsync(userId, tenantId) → main ingestion loop.
* OnboardOrganisationsAsync(userId, tenantId) → initial ingestion for new org.
* Helper methods for paging, retries, and mapping raw payloads into reports.

⚠️ **Error Handling & Validation**

* Wraps API failures in Result<T>.
* Catches exceptions and logs errors.
* Risks:
  + Partial ingestion (some endpoints fail) not clearly exposed in result.
  + Reliance on free-form error messages in logs.

🔐 **Security Review**

* Handles sensitive OAuth tokens via IXeroApiClient.
* Must avoid logging tokens or full payloads.
* Raw payloads may include PII/financial data → must be encrypted at rest.

⚡ **Performance & Reliability**

* Heavy DB + network I/O.
* Supports async and cancellation.
* Risks:
  + N+1 API calls if endpoints fetched individually.
  + Paging loops must be efficient and resilient.
* Should batch inserts for large payloads.

📊 **Observability**

* Logs ingestion start/end and per-endpoint results.
* Should provide metrics: rows inserted, errors, duration.

🧪 **Testability & Coverage**  
Suggested tests:

1. Successful ingestion across multiple endpoints.
2. API failure mid-run → partial success recorded.
3. Onboarding new org runs all required endpoints.
4. Cancellation halts ingestion.
5. Large payload handled correctly.

🧹 **Code Smells**

* **High:** Service is very large, mixes orchestration, logging, and persistence logic.
* **Medium:** Error reporting too coarse (single success/failure).
* **Low:** Overlaps with polling logic.

🔧 **Refactoring Suggestions**

* Extract endpoint ingestion into IEndpointIngestionService.
  + Effort: Large, Priority: 3.
* Replace free-form error strings with structured IngestionErrorSummary.
  + Effort: Medium, Priority: 3.
* Add retry policy abstraction (exponential backoff).
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Implements IDataIngestionService.
* Changes ripple across polling, onboarding, and ingestion handlers.

✅ **Confidence**  
High – robust, but too broad; needs decomposition for maintainability.

**File Review**

**File:** Application/Services/Organisation/OrganisationService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~450

🧾 **ELI5**  
This class manages organisations (tenants): fetching details, updating them, and assigning users to organisations with roles.

🎯 **Purpose and Role**

* Implements IOrganisationService.
* Provides CRUD-like functionality for organisations.
* Coordinates user–organisation–role relationships.
* Used in organisation commands/queries and onboarding flows.

🔍 **Detailed Breakdown**  
Dependencies:

* IOrganisationRepository → persistence of org records.
* IAccessService → checks permissions for assignments.
* ILogger<OrganisationService> → logs.
* IUnitOfWorkFactory → transactional consistency.

Key methods:

* GetInfoAsync(orgId) → returns OrganisationInfoDto.
* EditOrganisationAsync(orgId, name) → updates name.
* AssignUsersAsync(orgId, userIds) → bulk assigns users.
* GetUserOrganisationsAsync(userId) → returns user’s org memberships.
* EnsureUserOrganisationLinkAsync(orgId, userId, roleId, scopes) → ensures link exists.

⚠️ **Error Handling & Validation**

* Uses Result<T> wrappers.
* Handles null/missing org gracefully.
* Logs errors on failure.
* Risks:
  + Duplicate user assignments not explicitly prevented (depends on repo).
  + String-based scopes could allow invalid values.

🔐 **Security Review**

* Assignments must only be done by privileged users.
* Scopes stored as strings → risk of injection or misconfiguration.
* Logs must avoid sensitive role/scope details.

⚡ **Performance & Reliability**

* Bulk assignment loops must be efficient (batch DB writes).
* Queries must use indexes on OrgId + UserId.
* EnsureUserOrganisationLink should be idempotent.

📊 **Observability**

* Logs edits, assignments, and failures.
* Could benefit from structured logging (OrgId, UserId).

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid org retrieval → returns DTO.
2. Non-existent org → failure result.
3. Edit updates org name.
4. Assign multiple users works.
5. EnsureUserOrganisationLink prevents duplicates.
6. GetUserOrganisations returns correct set.

🧹 **Code Smells**

* **Medium:** Service mixes CRUD (org) and relationship (user-org-role).
* **Medium:** Scopes as raw strings instead of typed objects.

🔧 **Refactoring Suggestions**

* Split into OrganisationService (CRUD) and UserOrgService (assignments).
  + Effort: Medium, Priority: 3.
* Replace string-based scopes with value object or enum.
  + Effort: Medium, Priority: 4.

📜 **Contracts & Compatibility**

* Implements IOrganisationService.
* Changes ripple across organisation commands/queries and onboarding flows.

✅ **Confidence**  
High – solid, but suffers from scope-string and SRP issues.

**File Review**

**File:** Application/Services/Polling/PollingService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~300

🧾 **ELI5**  
This class runs scheduled polling jobs, fetching fresh data from Xero for organisations at the configured intervals.

🎯 **Purpose and Role**

* Implements IPollingService.
* Orchestrates polling for a user’s tenants.
* Used by scheduled background jobs.

🔍 **Detailed Breakdown**  
Dependencies:

* IDataIngestionService → executes ingestion.
* IPollingSettingsRepository → retrieves polling configs.
* ILogger<PollingService> → logging.

Key method:

* PollAsync(userId) → retrieves polling configs, loops tenants, triggers ingestion.

⚠️ **Error Handling & Validation**

* Catches exceptions and logs failures.
* Must handle:
  + Missing configs.
  + Token refresh errors.
  + Partial ingestion failures.

🔐 **Security Review**

* Uses userId to resolve tenants → must validate ownership.
* Must avoid logging sensitive token data.

⚡ **Performance & Reliability**

* Polling may trigger many ingestions → batching required.
* Risk of N+1 ingestion calls per tenant.
* Should support cancellation tokens.

📊 **Observability**

* Logs poll start/end and per-tenant failures.
* Metrics: number of orgs polled, duration, rows ingested.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid poll triggers ingestion.
2. Missing config → logs and skips.
3. Ingestion failure logged but does not halt others.
4. Cancellation stops mid-run.

🧹 **Code Smells**

* **Medium:** Tight coupling to ingestion service.
* **Low:** Only exposes one method, coarse granularity.

🔧 **Refactoring Suggestions**

* Split into TenantPollingService for finer-grained control.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Implements IPollingService.
* Contract affects job scheduler.

✅ **Confidence**  
High – clear but coarse-grained.

**File Review**

**File:** Application/Services/Polling/PollingSettingsService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~350

🧾 **ELI5**  
This class manages polling settings (frequency, run time) for organisations, storing and retrieving them from the repository.

🎯 **Purpose and Role**

* Implements IPollingSettingsService.
* Provides CRUD-like operations for polling schedules.
* Used by polling UI and scheduler.

🔍 **Detailed Breakdown**  
Dependencies:

* IPollingSettingsRepository → persistence.
* ILogger<PollingSettingsService> → logging.

Key methods:

* GetAsync(orgId) → settings for org.
* GetManyAsync(orgIds) → multiple orgs.
* GetAllAsync() → system-wide.
* UpsertAsync(orgId, frequency, runTime?) → create/update.

⚠️ **Error Handling & Validation**

* Returns Result<T> wrappers.
* Does not enforce frequency/runTime rules itself.
* Must handle invalid inputs (Hourly + runTime).

🔐 **Security Review**

* Not security-sensitive.
* Must validate tenant ownership on read/write.

⚡ **Performance & Reliability**

* Reads lightweight.
* Bulk queries must be efficient.
* Should be cached at scheduler startup.

📊 **Observability**

* Logs updates and retrievals.
* Should warn on invalid schedules.

🧪 **Testability & Coverage**  
Suggested tests:

1. Valid upsert persists config.
2. Invalid combo (Hourly + runTime) → failure.
3. Bulk retrieval returns correct map.
4. GetAll returns all records.

🧹 **Code Smells**

* **Medium:** Service adds little beyond repository.
* **Low:** Business rules deferred → risk of misuse.

🔧 **Refactoring Suggestions**

* Enforce frequency/runTime validation here.
  + Effort: Medium, Priority: 4.
* Add DeleteAsync(orgId).
  + Effort: Small, Priority: 3.

📜 **Contracts & Compatibility**

* Implements IPollingSettingsService.
* Breaking changes affect scheduler and dashboard.

✅ **Confidence**  
High – functional but thin abstraction.

✨ **Batch Summary**

* **PollingService** correctly orchestrates polling runs but is coarse-grained and tightly coupled.
* **PollingSettingsService** handles config persistence but pushes validation downstream.  
  Both work, but could be made more robust with stronger validation and finer-grained abstractions.

**File Review**

**File:** Application/Services/Scopes/XeroScopeService.cs  
**Layer/Type:** Application – Service Implementation  
**Status:** Reviewed  
**Tokens (approx.):** ~130

🧾 **ELI5**  
This class manages Xero OAuth scopes, fetching only those that are active and available for use in login and token flows.

🎯 **Purpose and Role**

* Implements IXeroScopeService.
* Retrieves active scopes from repository.
* Provides them to auth flows and token service.

🔍 **Detailed Breakdown**  
Dependencies:

* IXeroScopeRepository → persistence.
* ILogger<XeroScopeService> → logging.

Key method:

* GetActiveScopesAsync() → fetches active XeroScope list.

⚠️ **Error Handling & Validation**

* Returns Result<List<XeroScope>>.
* Logs errors if repository fails.
* Relies on repo to enforce uniqueness.

🔐 **Security Review**

* Scopes define system permissions.
* Must ensure only valid/active scopes are returned.
* Must not allow inactive/deprecated scopes into login flows.

⚡ **Performance & Reliability**

* Lightweight read operation.
* Should be cached for performance (scopes rarely change).

📊 **Observability**

* Logs errors when no scopes found or repo fails.

🧪 **Testability & Coverage**  
Suggested tests:

1. Active scopes exist → returned list.
2. No active scopes → empty list.
3. Repository error → failure result.

🧹 **Code Smells**

* **Low:** Service is thin wrapper around repository.

🔧 **Refactoring Suggestions**

* Add caching layer.
  + Effort: Small, Priority: 2.
* Add explicit validation for scope name format.
  + Effort: Medium, Priority: 3.

📜 **Contracts & Compatibility**

* Implements IXeroScopeService.
* Changes ripple into auth/token service.

✅ **Confidence**  
High – small, consistent, well-aligned with OAuth ADR.