**File Review: Infrastructure/Infrastructure.csproj**

**Layer/Type:** Infrastructure / Project File  
**Status:** Reviewed  
**Tokens:** ~250

**🧸 ELI5**

This file tells .NET how to build the **Infrastructure** project. It declares dependencies like **Dapper, Redis, Xero OAuth, Key Vault, logging, and email**, which power RoadmApp’s integration and persistence features.

**🎯 Purpose and Role**

* Defines the **Infrastructure layer’s responsibilities**:
  + Implements Application interfaces for DB, caching, external services.
  + Connects to **Xero OAuth2**, **Redis**, **Key Vault**, **SMTP (MailKit/MimeKit)**.
  + Provides persistence with **Dapper**.
* Consumed by Application layer (via interfaces) and Web layer (via DI).

**🔍 Detailed Breakdown**

* **Framework:** Targets net8.0.
* **ItemGroups:**
  + **References:**
    - Application/Application.csproj (respects Clean Architecture dependency direction).
  + **Folders:** Database, Email, Encryption, DependencyInjection, External/Xero, Scheduling, Locking.
  + **Packages:**
    - **Core support:** Microsoft.Extensions.Logging, Microsoft.Extensions.Http.Resilience, System.Text.Json.
    - **Auth & Security:** Microsoft.IdentityModel.JsonWebTokens, Xero.NetStandard.OAuth2, BCrypt.Net-Next.
    - **Data & Caching:** Dapper, Npgsql, Microsoft.Extensions.Caching.StackExchangeRedis, StackExchange.Redis.
    - **Messaging:** Azure.Messaging.ServiceBus.
    - **Email:** MailKit, MimeKit.
    - **Misc:** Polly (resilience), Scrutor (DI scanning).

**⚠️ Error Handling & Validation**

* Not applicable in project file, but presence of **Polly** suggests retries are baked into infra services.
* Ensure all Dapper queries use parameterisation (review in later DB code).

**🔐 Security Review**

* Good alignment with ADRs:
  + **Key Vault** implied by Encryption folder and packages.
  + **Xero OAuth2** integration supported by package references.
  + **BCrypt.Net** ensures password hashing done here (not in Application layer).
* ✅ No secrets hardcoded in project file.

**⚡ Performance & Reliability**

* **Dapper** chosen for lightweight DB operations (per ADR).
* **Redis** ensures distributed caching/scaling.
* **Polly** provides retries and resilience.
* **Service Bus** allows async event-driven processing.
* ✅ Aligned with architectural decisions.

**📊 Observability**

* Microsoft.Extensions.Logging dependency ensures structured logging.
* Expect correlation ID handling and centralised logging middleware in Web layer.

**🧪 Testability & Coverage**

* Infra is inherently harder to unit test (external integrations).
* Use **fakes or in-memory providers** for Redis, Service Bus, and SMTP in tests.
* Suggested test coverage:
  + Positive: Successful DB query via Dapper.
  + Negative: Simulated Redis outage with Polly retry.
  + Edge: Service Bus connection timeout.

**🛠 Code Smells**

* **Info:** Large number of package dependencies → risk of “god project” syndrome if responsibilities are not well-partitioned. Needs folder discipline.

**🔧 Refactoring Suggestions**

* **Quick Win (Priority 3):** Review necessity of **all included packages** (esp. Scrutor + manual DI, potential overlap).
* **Small (Priority 2):** Ensure unused folders (e.g., Scheduling, Locking) have meaningful contents or are removed to reduce clutter.

**📑 Contracts & Compatibility**

* No public APIs here, but **dependency versions** lock Infrastructure behaviour.
* Ensure consistent versioning with Application/Web .csproj to avoid mismatched assemblies.

**🗄 Data Model Notes**

* Npgsql confirms **PostgreSQL backend** per ADRs.
* Dapper chosen explicitly for performance reasons.

**✅ Confidence: High**

* Entire file was available and reviewed; strong alignment with Clean Architecture and ADRs.

**File Review: Infrastructure/DependencyInjection.cs**

**Layer/Type:** Infrastructure / Composition Root  
**Status:** Reviewed  
**Tokens:** ~1500

**🧸 ELI5**

This file wires up all the **Infrastructure services** (DB, Redis, Xero, email, encryption, logging, etc.) into .NET’s dependency injection system so the rest of the app can just “ask” for them without knowing how they’re built.

**🎯 Purpose and Role**

* Defines **one central entry point** for registering all Infrastructure dependencies.
* Follows Clean Architecture: Application layer depends on Infrastructure *via interfaces*, not vice versa.
* Provides runtime configuration bindings and enforces validation on required settings.

**🔍 Detailed Breakdown**

* **Extension method:** AddInfrastructureServices(IServiceCollection services, IConfiguration configuration, bool isTesting = false).
* **Configuration bindings with validation:**
  + XeroApiSettings → validates ClientId, Secret, CallbackUri, Base URLs.
  + SmtpSettings → validates Host, From, Port.
  + RoleSettings → validates AdminRoleId.
  + WebAppConfiguration → validates BaseUrl.
  + PythonEtlOptions → validates script paths.
* **Data Access:**
  + Registers IDbConnectionFactory → PostgresConnectionFactory.
  + Registers Dapper-based repositories (Token, Role, Permission, CallLog, Scopes, User, EndpointConfig, XeroScope, PasswordReset, Polling, etc.).
* **Caching & Locking:**
  + If Redis configured: uses DistributedTokenCache, RedisDistributedLock.
  + Else: defaults to in-memory implementations (MemoryTokenCache, InMemoryDistributedLock).
* **Security:**
  + Password hashing → BCryptPasswordHasher.
  + Token encryption → DataProtectionTokenEncryptionService.
* **Xero Integration:**
  + Registers XeroClient with resilient HTTP handler (Polly: max 3 retries, jitter backoff).
  + Wraps XeroApiClient in custom logging decorator (CallLoggingXeroApiClient).
* **Services:**
  + Adds scoped services: UserService, OrganisationService, PollingService, DashboardService, CallLogService, etc.
  + Registers hosted PollingScheduler for background ETL jobs.
* **Observability:**
  + Extensive use of logging (ILogger<> injected everywhere).
  + ServiceBus + WebhookQueue registration included.

**⚠️ Error Handling & Validation**

* Strong validation of configuration sections (throws on startup if missing/invalid).
* Polly retry for Xero API ensures resilience.
* Good default fallback when Redis is absent (in-memory).
* Guard clauses ensure AdminRoleId and critical URLs are not null.

**🔐 Security Review**

* ✅ Uses BCrypt.Net for password hashing (strong choice).
* ✅ Tokens encrypted with Data Protection.
* ✅ Secrets (Redis, SMTP, Xero credentials) bound via configuration, not hardcoded.
* ⚠️ **Observation:** Ensure that sensitive logs (tokens, passwords) are redacted before logging in CallLoggingXeroApiClient.

**⚡ Performance & Reliability**

* Redis caching + distributed locking ensures scalability across nodes (per ADR).
* Polly provides retry and jittered backoff for external calls (prevents cascading failure).
* ServiceBus integration allows async scaling.
* Potential bottleneck: synchronous DB operations if Dapper usage isn’t async.

**📊 Observability**

* Centralised logging wired everywhere.
* Background services (PollingScheduler) are host-registered → ensures visibility via hosted service logs.
* Retry attempts and circuit breaking logged by Polly.
* Correlation ID integration expected in middleware (not visible here).

**🧪 Testability & Coverage**

* **isTesting flag** allows bypassing of validation (good for test setups).
* Interfaces used extensively → easy mocking in unit tests.
* Suggested test coverage:
  + Positive: Startup with full valid config registers all services.
  + Negative: Missing SMTP or Redis section throws.
  + Edge: Redis misconfigured → falls back to in-memory.
  + Edge: Multiple concurrent token refreshes blocked by distributed lock.

**🛠 Code Smells**

* **Low:** This method is very large (~400+ lines). Harder to maintain.
* **Info:** In-memory fallbacks are useful, but risk accidental production use if Redis not configured.

**🔧 Refactoring Suggestions**

* **Medium Effort (Priority 4):** Split registration into **feature-specific extension methods** (e.g., AddXeroInfrastructure(), AddCachingInfrastructure(), AddPersistenceInfrastructure()). Keeps this file readable.
* **Quick Win (Priority 3):** Add explicit logging when falling back to in-memory cache/lock, to alert operators in distributed deployments.
* **Small (Priority 2):** Consider enabling health check registrations here (e.g., AddHealthChecks().AddNpgSql().AddRedis()) to make infra self-validating.

**📑 Contracts & Compatibility**

* Defines **runtime contract** between Infra and Application via interfaces.
* Changing registered services here could break MediatR handlers if interfaces mismatch.
* Must preserve Dapper + Redis ADR strategy.

**🗄 Data Model Notes**

* DB layer bound via PostgresConnectionFactory → confirms **PostgreSQL is the primary backend**.
* Repositories abstract queries, leaving schema changes outside this file’s scope.

**✅ Confidence: High**

* Full file available. Dependency wiring clearly aligns with ADRs and Clean Architecture.

**File Review: Infrastructure/Cache/DistributedTokenCache.cs & MemoryTokenCache.cs**

**Layer/Type:** Infrastructure / Caching  
**Status:** Reviewed  
**Tokens:** ~1000 combined

**🧸 ELI5**

These files are the app’s **“token cupboards.”**

* DistributedTokenCache: Stores Xero OAuth2 tokens in **Redis** so all servers in a cluster can share them.
* MemoryTokenCache: Stores tokens **in-memory** for local or single-instance deployments.

**🎯 Purpose and Role**

* Both implement ITokenCache, the Application interface for token storage.
* Ensures that OAuth2 tokens (access + refresh) are cached efficiently, following ADR strategy.
* Used by the XeroTokenService for rotation and refresh handling.

**🔍 Detailed Breakdown**

**DistributedTokenCache**

* Uses IDistributedCache (backed by Redis).
* Serialises/deserialises XeroOAuth2Token to JSON.
* Methods:
  + TryGet(Guid userId, out XeroOAuth2Token?): Attempts to load token.
  + Set(Guid userId, XeroOAuth2Token token): Writes token with expiry (default 90 days).
  + Remove(Guid userId): Deletes from cache.
* Logs warnings on cache failures.

**MemoryTokenCache**

* Uses thread-safe ConcurrentDictionary<Guid, XeroOAuth2Token>.
* Methods mirror the distributed version: TryGet, Set, Remove.
* Provides fast, in-process caching but not shared across nodes.

**⚠️ Error Handling & Validation**

* Both handle **cache miss** cleanly by returning false instead of throwing.
* Distributed version catches exceptions (e.g., Redis unavailable) and logs warnings.
* Memory version assumes in-process safety with ConcurrentDictionary.

**🔐 Security Review**

* ✅ Tokens are serialised, not logged.
* ✅ Cache keys are prefixed (xero:token:{userId}) to avoid collisions.
* ⚠️ Ensure **token JSON** does not leak sensitive values into logs if serialization errors occur.
* ✅ Redis fallback strategy already defined in ADR (memory cache for single node, Redis for multi-node).

**⚡ Performance & Reliability**

* **Distributed:** Scales across nodes; Redis TTL ensures stale tokens eventually expire.
* **Memory:** Extremely fast, but only safe for single-instance deployments.
* ✅ Aligns with ADRs: no extra custom caching strategy introduced.

**📊 Observability**

* Distributed cache logs warnings on failures.
* Memory cache silent → may make debugging harder (e.g., token unexpectedly missing).

**🧪 Testability & Coverage**

Suggested tests:

1. **Happy path:** Store and retrieve a token for a user.
2. **Negative:** Retrieve before storing → should return false.
3. **Edge:** Remove non-existent key → should not throw.
4. **Distributed edge:** Simulate Redis unavailable → should log warning and fail gracefully.
5. **Expiry edge:** Token expires in Redis → ensure it cannot be retrieved.

**🛠 Code Smells**

* **Info:** DistributedTokenCache hardcodes default expiry (90 days). Should be configurable.
* **Low:** Lack of logging in MemoryTokenCache → makes token lifecycle opaque.
* **Medium:** Both caches duplicate method signatures. Could extract shared base to reduce duplication.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Add structured logging in MemoryTokenCache for Set and Remove.
* **Small (Priority 2):** Move cache expiry config to appsettings.json or XeroApiSettings instead of hardcoding.
* **Medium (Priority 2):** Extract common interface implementation (e.g., BaseTokenCache) to reduce duplication.

**📑 Contracts & Compatibility**

* Implements ITokenCache → contract used in Application layer.
* Must remain backward compatible; changing key format or expiry rules would break token refresh flow.

**🗄 Data Model Notes**

* Cache is key-value only. No relational model involved.
* Redis implementation leverages DistributedCacheEntryOptions for expiry.

**✅ Confidence: High**

* Both full files were available and reviewed; behaviour is clear and aligns with ADR token strategy.

**File Review: Infrastructure/Encryption/DataProtectionTokenEncryptionService.cs**

**Layer/Type:** Infrastructure / Security  
**Status:** Reviewed  
**Tokens:** ~800

**🧸 ELI5**

This file is like a **safe for secrets**: it encrypts and decrypts sensitive tokens (like Xero OAuth2 tokens) using .NET’s built-in **Data Protection API**, so tokens are never stored or cached in plain text.

**🎯 Purpose and Role**

* Implements ITokenEncryptionService from Application layer.
* Provides secure token handling using IDataProtector.
* Ensures Application and Domain layers don’t deal with raw crypto — Infrastructure encapsulates it.

**🔍 Detailed Breakdown**

* **Constructor:** Requires IDataProtectionProvider, creates a protector with the purpose "XeroTokenProtection".
* **Methods:**
  + EncryptAsync(string plaintext, CancellationToken)
    - Validates input non-null.
    - Uses \_protector.Protect(plaintext).
    - Returns encrypted string wrapped in Task.
  + DecryptAsync(string ciphertext, CancellationToken)
    - Validates input non-null.
    - Uses \_protector.Unprotect(ciphertext).
    - Returns decrypted string wrapped in Task.
* **Design notes:**
  + Throws ArgumentNullException for null/empty input.
  + Wraps Data Protection’s synchronous API in async-friendly Task.FromResult.

**⚠️ Error Handling & Validation**

* ✅ Proper null checks before encrypt/decrypt.
* ❌ Does not catch CryptographicException during decrypt → unhandled corruption could bubble up.
* Would benefit from **try/catch → log → return failure result** instead of throwing.

**🔐 Security Review**

* ✅ Uses ASP.NET Core Data Protection → strong, enterprise-grade crypto.
* ✅ Avoids custom crypto implementation.
* ✅ Purpose string ensures scope isolation ("XeroTokenProtection").
* ✅ Encryption service stays in Infrastructure layer (per ADR).
* ⚠️ Ensure persistence of Data Protection keys is configured correctly (Key Vault / storage) in Web layer, else encryption may break across deployments.

**⚡ Performance & Reliability**

* Lightweight → single method call around Data Protection.
* Task.FromResult avoids unnecessary thread pool usage.
* Risk: If Data Protection keys rotate or are lost, previously encrypted tokens become unreadable.

**📊 Observability**

* No logging currently → if decryption fails, only exception surfaces.
* Suggest logging warnings for failed decrypt attempts (without exposing sensitive values).

**🧪 Testability & Coverage**

Suggested tests:

1. **Happy path:** Encrypt a token → decrypt it → should return original.
2. **Negative:** Pass null or empty → should throw ArgumentNullException.
3. **Edge:** Decrypt invalid ciphertext → should throw CryptographicException.
4. **Edge:** Encrypt very large token string (e.g., >4KB).

**🛠 Code Smells**

* **Low:** Lacks structured error handling/logging in DecryptAsync.
* **Info:** Async signatures return Task.FromResult — valid but might mislead; could consider ValueTask.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Add try/catch in DecryptAsync with warning log.
* **Quick Win (Priority 2):** Replace Task.FromResult with ValueTask for slightly more idiomatic async usage.
* **Medium (Priority 2):** Ensure key persistence configured via DI setup (probably in Web layer).

**📑 Contracts & Compatibility**

* Implements Application interface → do not change method signatures.
* If encryption purpose string changes, old tokens become undecryptable → breaking change.

**🗄 Data Model Notes**

* Not applicable (stateless service).

**✅ Confidence: High**

* Full file reviewed; design matches ADRs and secure cryptography best practices.

**File Review: Infrastructure/External/Xero/XeroApiClient.cs**

**Layer/Type:** Infrastructure / External Integration (Xero API)  
**Status:** Reviewed  
**Tokens:** ~3,000

**🧸 ELI5**

This file is the **bridge between RoadmApp and Xero**. It handles talking to Xero’s API: logging in, refreshing tokens, fetching connections, invoices, and handling tenant-specific requests.

**🎯 Purpose and Role**

* Implements IXeroApiClient (Application-facing contract).
* Encapsulates **all raw HTTP calls to Xero APIs**.
* Handles authentication (Bearer tokens, tenant IDs), retries, logging, and error translation.
* Used by XeroTokenService and higher-level features in the Application layer.

**🔍 Detailed Breakdown**

* **Constructor:** Requires HttpClient, IOptions<XeroApiSettings>, and ILogger<XeroApiClient>.
  + Validates ClientId, ClientSecret, CallbackUri, BaseUrl.
  + Configures default headers to accept JSON.
* **Methods:**
  + GetConnectionsAsync() → Fetches tenant connections for a given access token.
  + RefreshTokenAsync() → Exchanges refresh token for new access + refresh token pair.
  + FetchDataForEndpointAsync() → General method for retrieving endpoint data (Invoices, etc.). Supports If-Modified-Since.
  + BuildLoginUri() → Creates Xero OAuth2 login URL with scopes and redirect URI.
  + GetAccessTokenAsync() → Exchanges authorization code for access token.
  + GetInvoiceAsync() → Fetches invoice JSON for given tenant and invoice ID.
  + DeleteConnectionAsync() → Disconnects a Xero tenant.
* **Private helper:** ApplyDefaultHeaders() ensures every request has proper headers (Authorization, Accept, xero-tenant-id, optional Idempotency-Key).

**⚠️ Error Handling & Validation**

* ✅ Validates null/empty parameters (e.g., tenantId, authCode).
* ✅ Logs errors/warnings for failed HTTP responses, including status codes and bodies.
* ✅ Throws AuthenticationException when refresh tokens are invalid.
* ❌ Some exception blocks rethrow without wrapping → may bubble raw exceptions to Application layer.
* ❌ JSON deserialization errors logged but method returns null instead of a failure object.

**🔐 Security Review**

* ✅ No secrets hardcoded; clientId and clientSecret injected via config.
* ✅ Tokens only used in Authorization header.
* ✅ Tenant ID passed in headers, not in query string.
* ⚠️ Logs include **response bodies** on error — risk of exposing sensitive PII (invoices, contact info). Should sanitize before logging.
* ✅ Aligns with ADR: OAuth2 token strategy & tenant separation.

**⚡ Performance & Reliability**

* Uses HttpClient properly (injected, not recreated).
* Supports conditional requests with If-Modified-Since → reduces unnecessary fetches.
* Token refresh and rotation handled externally (delegated to XeroTokenService).
* ❌ Potential issue: no retry logic directly in this class (Polly applied in DI setup, so this is acceptable).
* Large responses (Invoices, etc.) fully deserialized → consider streaming for big payloads.

**📊 Observability**

* Extensive logging at error/warning/debug levels.
* Logs URIs, status codes, response bodies.
* ✅ Good for debugging.
* ⚠️ Risk of **PII in logs** → should truncate or mask sensitive JSON fields.

**🧪 Testability & Coverage**

* Heavily dependent on HttpClient. Should be tested with **mocked HttpMessageHandler**.
* Suggested tests:
  1. Positive: Fetch tenant connections with valid token.
  2. Negative: Invalid refresh token → throws AuthenticationException.
  3. Edge: Empty connections list returns empty result.
  4. Error handling: Invalid JSON response → logged + returns null.
  5. Edge: Large invoice payload handled without memory issues.

**🛠 Code Smells**

* **High:** Logs may expose raw Xero API responses (security/privacy risk).
* **Medium:** Class size is very large (~18KB) → mixing many responsibilities.
* **Low:** Task<string?> GetInvoiceAsync returns raw JSON string → inconsistent with DTO usage elsewhere.

**🔧 Refactoring Suggestions**

* **Large Effort (Priority 5):** Split into feature-specific clients (e.g., XeroAuthClient, XeroInvoiceClient, XeroConnectionClient) for maintainability.
* **Small Effort (Priority 4):** Introduce response sanitization before logging.
* **Medium Effort (Priority 3):** Wrap JSON parsing in a consistent Result<T> instead of returning null.
* **Quick Win (Priority 2):** Use ValueTask for lightweight operations like BuildLoginUri().

**📑 Contracts & Compatibility**

* Implements IXeroApiClient → Application layer expects stability.
* Login URI and token exchange follow OAuth2 spec — breaking changes would disrupt Xero onboarding.
* GetInvoiceAsync returning raw JSON ties Application logic to Xero schema. If Xero API changes, downstream may break.

**🗄 Data Model Notes**

* No internal DB persistence here — this class only retrieves/parses API responses.
* Downstream mapping handled in Mappers/.

**✅ Confidence: High**

* Full file reviewed. Class is central to Xero integration and tightly aligned with ADRs.

**File Review: Infrastructure/External/Xero/XeroAuthHandler.cs**

**Layer/Type:** Infrastructure / Authentication Middleware  
**Status:** Reviewed  
**Tokens:** ~800

**🧸 ELI5**

This file is a **gatekeeper for Xero API requests**. Before any HTTP call goes out, it injects the right user’s **access token** and **tenant ID** into the headers so Xero knows who’s asking.

**🎯 Purpose and Role**

* Implements a DelegatingHandler for HttpClient.
* Ensures every Xero API request has a valid OAuth2 access token and correct tenant ID.
* Uses IXeroTokenService to fetch/refresh tokens as needed.
* Ties into DI pipeline when registering HttpClient for Xero.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IXeroTokenService → fetches valid tokens.
  + ILogger<XeroAuthHandler> → structured logging.
  + IHttpContextAccessor → retrieves current user & tenant context.
* **Core logic:** SendAsync(HttpRequestMessage request, CancellationToken ct)
  + Retrieves ICurrentUserContext from request scope.
  + Validates UserId and TenantId. Logs error + returns 401 if missing.
  + Uses IXeroTokenService.GetValidTokenAsync() to retrieve token.
  + If token unavailable → logs error + returns 401.
  + Calls SetRequestHeaders() → injects Authorization: Bearer {token} and xero-tenant-id.
  + Proceeds with base.SendAsync(request, ct).
  + If response = 401 Unauthorized → logs warning (token likely expired).
* **Helpers:**
  + SetRequestHeaders() → ensures Authorization + tenant ID headers are set without duplication.
* **User context:**
  + Defines ICurrentUserContext interface (UserId, TenantId).
  + Implements CurrentUserContext with SetContext(Guid userId, Guid tenantId).

**⚠️ Error Handling & Validation**

* ✅ Handles missing context with explicit 401.
* ✅ Logs meaningful errors if token not found.
* ✅ Correctly distinguishes between missing User/Tenant and expired tokens.
* ❌ Relies on IXeroTokenService to refresh — if that fails, handler just returns 401 without retry.

**🔐 Security Review**

* ✅ Tokens only injected into headers, never logged.
* ✅ Prevents duplicate tenant headers by clearing before adding.
* ✅ 401 returned if context invalid, avoiding accidental “open” requests.
* ⚠️ Risk: If logging is misconfigured, tenant/user IDs could be exposed in logs. Masking may be needed for compliance.

**⚡ Performance & Reliability**

* Lightweight — adds minimal overhead per request.
* Uses scoped DI context, so safe for multi-user scenarios.
* ❌ Single retry gap: if a token has just expired, user may see 401 instead of automatic refresh (but ADR places refresh responsibility in XeroTokenService, not here).

**📊 Observability**

* Good logging coverage:
  + Errors when context missing or token invalid.
  + Warnings on 401 responses.
* Logging includes request URI and user/tenant IDs → helpful for debugging, but risk of oversharing.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Valid context and token → request succeeds with headers.
2. Negative: Missing UserId/TenantId → returns 401.
3. Negative: Token not found in service → returns 401.
4. Edge: Token valid but downstream returns 401 → should log warning.
5. Edge: Duplicate tenant header not injected twice.

**🛠 Code Smells**

* **Low:** Handler both validates context and manages headers → could be split for SRP.
* **Info:** CurrentUserContext set manually → risk of being forgotten in some request pipelines.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Add unit tests for SetRequestHeaders to ensure no duplication.
* **Small (Priority 2):** Mask user/tenant IDs in logs unless debugging.
* **Medium (Priority 3):** Consider moving 401 retry logic (token refresh) into handler to improve UX (but must align with ADR token refresh design).

**📑 Contracts & Compatibility**

* Must remain consistent with IXeroTokenService contract.
* Any change to header injection risks breaking Xero API requests.
* Contractually ensures tenant separation (multi-tenancy enforced at request level).

**🗄 Data Model Notes**

* Not applicable (transient per-request handler).

**✅ Confidence: High**

* Full file reviewed. Clear adherence to ADRs and good layering.

**File Review: Infrastructure/External/Xero/XeroTokenService.cs**

**Layer/Type:** Infrastructure / External Integration (Token Orchestration)  
**Status:** Reviewed  
**Tokens:** ~2,800

**🧸 ELI5**

This file is the **token manager** for Xero. It makes sure RoadmApp always has a valid access token: creating new ones when users log in, refreshing them when they expire, rotating them every 55 days, and securely storing them.

**🎯 Purpose and Role**

* Implements IXeroTokenService.
* Core responsibilities:
  + Exchange auth codes for tokens.
  + Refresh access tokens before expiry.
  + Rotate refresh tokens after 55 days.
  + Cache tokens (Redis/in-memory).
  + Encrypt tokens before persisting.
  + Clean up (disconnect tenants, purge tokens).
* Works with XeroApiClient, ITokenRepository, ITokenCache, and distributed locking.
* Follows ADR-defined OAuth2 strategy.

**🔍 Detailed Breakdown**

* **Constructor Dependencies:**
  + ITokenRepository → database persistence.
  + ITokenCache → Redis/memory cache.
  + IXeroApiClient → raw API calls.
  + ITokenEncryptionService → secure storage.
  + ILogger<XeroTokenService> → logging.
  + XeroApiSettings → clientId, secret, redirect URIs.
  + XeroClient → external SDK client.
  + IDistributedLock → ensures only one refresh per user at a time.
* **Constants:**
  + \_refreshWindow = 5 minutes (refresh before expiry).
  + \_rotationWindow = 55 days (rotate refresh tokens).
* **Key Methods:**
  + AuthorizeAsync(code, userId) → exchanges code for token, stores encrypted in DB + cache.
  + GetValidTokenAsync(userId) → retrieves token, refreshes if expiring or due for rotation.
  + RetrieveTokenAsync(userId) → gets from cache, else DB.
  + TryRefreshTokenAsync(token, userId) → refreshes via API, persists new values.
  + StoreTokenAsync(token, userId) → encrypts, persists, caches.
  + DestroyTokenForUserAsync(userId) → deletes all tokens for user.
  + RemoveTenantAsync(userId, tenantId) → removes tenant from user’s token.
  + GetUserIdForTenantAsync(tenantId) → reverse lookup tenant → user.
  + DisconnectTenantAsync(userId, tenantId) → API + repo removal.

**⚠️ Error Handling & Validation**

* ✅ Extensive try/catch blocks with logging on token refresh failures.
* ✅ Logs informative messages (expiry, rotation, cache hits).
* ✅ Uses distributed lock to prevent duplicate refreshes.
* ✅ Returns Result<T> where appropriate, making failures explicit.
* ❌ Some failure paths return null (e.g., decrypt failure) → could be made explicit with Result.Failure.

**🔐 Security Review**

* ✅ Encrypts tokens at rest (ITokenEncryptionService).
* ✅ Passwords never stored here (uses only OAuth2).
* ✅ Redis/memory caches store JSON-serialized tokens only in volatile storage.
* ✅ Logs avoid printing raw tokens.
* ⚠️ **Minor risk:** Tenant/user IDs logged; may need masking if PII policies apply.
* ✅ ADR compliance: refresh window (5 mins), rotation window (55 days).

**⚡ Performance & Reliability**

* ✅ Redis cache avoids DB hits.
* ✅ Distributed locks prevent race conditions on refresh.
* ✅ Refresh logic jittered (500ms delay before retry).
* ❌ DB round-trips may become heavy under many tenants → could benefit from batching in future.
* ✅ Failures gracefully logged and token purged.

**📊 Observability**

* ✅ Logs debug/info/warning/error events for token lifecycle.
* ✅ Clear messages when tokens refreshed, rotated, removed.
* ❌ No explicit metrics (e.g., refresh success/failure counts). Would help in ops.

**🧪 Testability & Coverage**

Suggested tests:

1. **Happy path:** Valid token retrieved from cache.
2. **Negative:** Token missing → returns null.
3. **Edge:** Token expires in <5 mins → refresh triggered.
4. **Edge:** Token older than 55 days → rotation triggered.
5. **Concurrency:** Two parallel refreshes → distributed lock ensures only one runs.
6. **Failure:** Decryption fails → returns null + logs warning.

**🛠 Code Smells**

* **Medium:** Class is very large (~500+ lines). Hard to maintain.
* **Low:** Some methods return null instead of Result.Failure. Inconsistent contract.
* **Info:** Tenant removal logic duplicates cache + repo update patterns.

**🔧 Refactoring Suggestions**

* **Large Effort (Priority 5):** Split into smaller services (e.g., TokenRefreshService, TokenPersistenceService). Improves maintainability.
* **Medium (Priority 4):** Replace null returns with Result.Failure for consistency.
* **Small (Priority 3):** Add Prometheus/OpenTelemetry counters for refresh attempts/failures.
* **Quick Win (Priority 2):** Standardize logging templates (mask tenant IDs if sensitive).

**📑 Contracts & Compatibility**

* Implements IXeroTokenService (Application depends on it).
* Changing method signatures would break downstream handlers.
* Must maintain refresh/rotation contract as per ADRs (5 mins, 55 days).

**🗄 Data Model Notes**

* Token records persisted via ITokenRepository in DB.
* Tenant mapping stored inside token entity (TenantId, TenantName).

**✅ Confidence: High**

* Full file reviewed. Strong alignment with ADR token management strategy.

**File Review: Infrastructure/External/Xero/Decorators/XeroTokenEnrichingDecorator.cs**

**Layer/Type:** Infrastructure / External Integration (Decorator)  
**Status:** Reviewed  
**Tokens:** ~850

**🧸 ELI5**

This file is like a **translator** that makes sure Xero tokens include all the details RoadmApp needs (like tenant info and expiry times) before they’re stored or used.

**🎯 Purpose and Role**

* Implements **Decorator pattern** for IXeroTokenService.
* Wraps an existing IXeroTokenService to **enrich tokens** with extra metadata (e.g., tenant info, expiry normalization).
* Ensures that every token leaving the service is complete and consistent for Application/Domain layers.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IXeroTokenService \_inner → the wrapped real service.
  + IXeroApiClient \_apiClient → fetches tenant info.
  + ILogger<XeroTokenEnrichingDecorator> → structured logging.
* **Enrichment Logic:**
  + On calls like GetValidTokenAsync, RetrieveTokenAsync, AuthorizeAsync, it fetches the token from \_inner.
  + If token is not null, attempts to enrich it with:
    - Latest tenant connections from Xero API.
    - Normalized expiry/rotation timestamps.
  + If enrichment fails → logs warning but still returns the base token.
* **Methods Overridden:**
  + AuthorizeAsync() → adds enrichment after token exchange.
  + GetValidTokenAsync() → enriches refreshed tokens.
  + RetrieveTokenAsync() → enriches cached/persisted tokens.
  + Other token lifecycle methods simply delegate to \_inner.

**⚠️ Error Handling & Validation**

* ✅ Catches exceptions during enrichment, logs warning, continues with base token.
* ✅ Prevents app from crashing due to enrichment failure.
* ❌ If enrichment consistently fails (e.g., Xero API down), tokens may lack tenant info but still circulate.

**🔐 Security Review**

* ✅ No secrets exposed; tokens are passed through unaltered except for enrichment.
* ✅ Tenant info enrichment does not log sensitive values.
* ⚠️ Logs may include tenant IDs/names; should ensure masking in sensitive environments.

**⚡ Performance & Reliability**

* Adds an **extra API call to Xero** during enrichment → may increase latency.
* Safe fallback (logs warning + continues) ensures reliability.
* Could result in **stale tenant info** if enrichment skipped.

**📊 Observability**

* Logs enrichment failures at warning level.
* Does not currently emit metrics (e.g., “% of tokens enriched”).

**🧪 Testability & Coverage**

Suggested tests:

1. **Positive:** Authorize token → enrichment adds tenant info.
2. **Negative:** API enrichment fails → token still returned, log warning.
3. **Edge:** Token already has tenants → ensure not duplicated.
4. **Edge:** Tenant list empty → token returns with no tenants.

**🛠 Code Smells**

* **Low:** Adds conditional enrichment logic in multiple methods → duplication risk.
* **Info:** Decorator logs warnings but doesn’t surface enrichment errors to caller.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Extract enrichment logic into private helper method to reduce duplication.
* **Medium (Priority 3):** Add metrics for enrichment success/failure rate.
* **Quick Win (Priority 2):** Consider caching tenant info to avoid hitting Xero API repeatedly.

**📑 Contracts & Compatibility**

* Fully compatible with IXeroTokenService contract.
* Safe wrapper: callers don’t need to know whether enrichment succeeded.
* Must remain aligned with Domain entity XeroOAuth2Token enrichment fields.

**🗄 Data Model Notes**

* Tenant enrichment aligns with XeroTenantInfo domain entity.
* No persistence here — token saving still handled by inner service.

**✅ Confidence: High**

* Full file reviewed. Proper Decorator usage, minimal risk, good alignment with ADR token model.

**File Review: Infrastructure/External/Xero/Mappers/SdkTokenMapper.cs**

**Layer/Type:** Infrastructure / Mapping (Xero SDK ↔ Domain)  
**Status:** Reviewed  
**Tokens:** ~1,000

**🧸 ELI5**

This file is a **converter**: it takes Xero’s SDK tokens (from their library) and turns them into RoadmApp’s internal **Domain token models**, and vice versa, so the rest of the app doesn’t need to understand Xero’s format.

**🎯 Purpose and Role**

* Bridges between external **Xero SDK (XeroOAuth2Token)** and RoadmApp’s **Domain entities**.
* Ensures consistency of token storage, encryption, and rotation timestamps.
* Called by XeroTokenService when persisting tokens or exchanging them.

**🔍 Detailed Breakdown**

* **Methods:**
  + ToDomainModel(SdkToken, tenants) → Maps Xero SDK token to RoadmApp’s XeroOAuth2Token.
    - Sets access token, refresh token, expiry times, and attached tenant info.
  + ToSdkToken(DomainToken) → Maps back to Xero SDK format.
  + FromRecords(List<TokenRecord>, ITokenEncryptionService) → Rebuilds domain tokens from DB records, decrypting sensitive values.
  + ToRecords(XeroOAuth2Token, userId, ITokenEncryptionService) → Converts domain token to one or more DB records for persistence, encrypting values.
* **Handling of Dates:**
  + Normalizes UTC expiry/rotation timestamps.
  + Ensures refresh issue times are tracked for 55-day rotation rule.
* **Tenant Info:**
  + Tokens enriched with XeroTenantInfo list.
  + Maps Xero’s connection data into domain structure.

**⚠️ Error Handling & Validation**

* ✅ Throws ArgumentNullException when mandatory parameters missing.
* ✅ Gracefully handles empty tenant lists (token still valid, just no tenants).
* ❌ Minimal logging in mapping functions → hard to diagnose failures during encryption/decryption.

**🔐 Security Review**

* ✅ Sensitive fields (AccessToken, RefreshToken) always passed through ITokenEncryptionService when persisting.
* ✅ Decrypts only when needed in-memory.
* ✅ No tokens written to logs.
* ⚠️ Risk: If encryption service misconfigured, tokens could be persisted in plaintext — relies on correct DI setup.

**⚡ Performance & Reliability**

* Lightweight mapping operations.
* LINQ used for record transformations — efficient enough for typical tenant counts.
* Reliability depends on ITokenEncryptionService. Failures bubble up but no retry.

**📊 Observability**

* No logging inside mapping functions. Failures rely on calling service to catch/log.
* Would benefit from debug-level logging when tokens are persisted/reconstructed.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Map SDK token → domain model with tenants.
2. Negative: Null SDK token → ArgumentNullException.
3. Edge: Token with empty tenant list → still valid domain token.
4. Round-trip: Map domain token → records → back to domain token (encrypted/decrypted correctly).
5. Security: Verify access/refresh tokens never persisted in plaintext.

**🛠 Code Smells**

* **Low:** Logging absent — debugging issues in token persistence may be difficult.
* **Info:** Mapper handles both conversion and encryption concerns → coupling could be reduced.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Add logging around encryption/decryption operations.
* **Medium (Priority 3):** Consider splitting mapping vs. persistence concerns into separate classes (TokenMapper vs. TokenPersistenceMapper).
* **Quick Win (Priority 2):** Add unit tests for round-trip mapping integrity.

**📑 Contracts & Compatibility**

* Must remain aligned with Domain entity XeroOAuth2Token and DB schema via TokenRecord.
* Changing mapping logic risks breaking token persistence/refresh lifecycle.

**🗄 Data Model Notes**

* DB persistence occurs via TokenRecord → likely table with UserId, EncryptedToken, Expiry.
* Mapper enforces ADR encryption at persistence boundary.

**✅ Confidence: High**

* Full file reviewed. Strong alignment with ADR token handling strategy.

**File Review: Infrastructure/External/Xero/Services/CallLoggingXeroApiClient.cs**

**Layer/Type:** Infrastructure / External Integration (Decorator/Logging)  
**Status:** Reviewed  
**Tokens:** ~1,100

**🧸 ELI5**

This file is like a **security camera** for the Xero API. It wraps around the main client and logs every API call RoadmApp makes to Xero, recording whether it succeeded, failed, or was throttled.

**🎯 Purpose and Role**

* Implements IXeroApiClient.
* Wraps a “real” Xero client and intercepts calls to log them via ICallLogService.
* Ensures all Xero API interactions are auditable in RoadmApp’s database.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IXeroApiClient \_inner → actual API implementation.
  + ICallLogService \_callLogService → persists logs of API calls.
  + ILogger<CallLoggingXeroApiClient> → structured logging.
* **Methods Overridden:**
  + BuildLoginUri(), GetAccessTokenAsync(), GetConnectionsAsync(), RefreshTokenAsync(),  
    FetchDataForEndpointAsync(), GetInvoiceAsync(), DeleteConnectionAsync().
  + Each delegates to \_inner but wrapped in try/catch.
* **Logging:**
  + On **success:** logs tenantId, endpoint, status, number of rows (if available).
  + On **failure:** logs exception and persists error details in DB.
  + Uses ICallLogService.LogCallAsync(...) with timestamp, tenant, endpoint name, status, error details.

**⚠️ Error Handling & Validation**

* ✅ All calls wrapped in try/catch.
* ✅ Logs both success and failure with context.
* ❌ Still rethrows exceptions → caller must handle them separately.
* ❌ If logging fails (DB error), API call result still returned, but logging gap created.

**🔐 Security Review**

* ✅ Tokens never logged.
* ✅ Logs contain tenantId + endpoint names (expected for audit).
* ⚠️ Potential risk: logs may store **raw error messages** from Xero, which could contain sensitive data. Should sanitize before persisting.

**⚡ Performance & Reliability**

* Adds logging overhead to each call, but async.
* Risk: Logging failures could impact performance if ICallLogService is slow.
* No retries for failed logs — could result in audit gaps.

**📊 Observability**

* ✅ Every Xero API call is observable.
* ✅ Success/failure rates trackable via logs.
* ❌ No metrics integration (e.g., Prometheus counters).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Successful API call logged with correct status.
2. Negative: API call fails → exception logged and rethrown.
3. Edge: API call returns 404 → logged as failure with details.
4. Edge: Logging service fails → API call result still returned.

**🛠 Code Smells**

* **Low:** Some duplication in logging patterns across methods.
* **Info:** Tight coupling to ICallLogService; consider making logging async-fire-and-forget.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Extract common try/catch + log pattern into private helper.
* **Small (Priority 3):** Add metrics (success/failure counters).
* **Quick Win (Priority 2):** Sanitize raw Xero error responses before logging to DB.

**📑 Contracts & Compatibility**

* Must remain compatible with IXeroApiClient contract.
* Breaking change to method signatures would disrupt Application handlers.
* Logging side effects must not alter API call results.

**🗄 Data Model Notes**

* Logs persisted via CallLogEntry DTO → stored in DB for later retrieval.
* Must ensure schema accommodates error details without truncation.

**✅ Confidence: High**

* Full file reviewed. Decorator pattern well-applied, aligns with ADR for observability.

**File Review: Infrastructure/External/Xero/Services/PythonEtlBridge.cs**

**Layer/Type:** Infrastructure / External Integration (ETL Orchestration)  
**Status:** Reviewed  
**Tokens:** ~1,200

**🧸 ELI5**

This file is like a **remote control** that tells Python scripts to run ETL jobs. RoadmApp starts an external Python process, passes the right parameters, waits for it to finish, and captures the logs and JSON output.

**🎯 Purpose and Role**

* Implements IPythonEtlBridge.
* Runs external Python ETL scripts to load Xero data into the database.
* Provides standardised orchestration around script execution, logging, error handling, and JSON summary output.
* Bridges Infrastructure (.NET) with existing Python ETL workflows.

**🔍 Detailed Breakdown**

* **Constructor:**
  + Reads default connection string from config.
  + Injects IOptionsMonitor<PythonEtlOptions> → per-profile configuration (script path, working directory, template directory).
  + Injects ILogger<PythonEtlBridge>.
* **Method:** RunAsync(profileName, tenantId, userId, batchId, sourceNames, cancellationToken)
  + Fetches ETL options for profile. Validates script path and enabled flag.
  + Builds ProcessStartInfo for Python interpreter + script.
    - Passes arguments: --stage profileName --tenant tenantId --user userId --batch batchId.
  + Sets environment variables:
    - ETL\_PG\_CONN = DB connection string.
    - ETL\_DEBUG\_MODE, ETL\_STOP\_ON\_ERROR, ETL\_TEMPLATE\_DIR, ETL\_MODELS\_DIR.
    - ETL\_SOURCES = comma-separated list of sources.
    - PYTHONPATH configured if needed.
  + Launches process, streams stdout/stderr into StringBuilder.
  + Waits for completion (with cancellation support).
  + On success: logs output, parses last JSON line for summary.
  + On failure: logs stderr, throws InvalidOperationException.
* **Outputs:**
  + JSON summary string if script produced valid JSON.
  + Null if ETL disabled.

**⚠️ Error Handling & Validation**

* ✅ Validates script path and enabled flag.
* ✅ Captures both stdout and stderr.
* ✅ Logs warnings if JSON summary not valid.
* ❌ Throws raw exceptions for failure (not wrapped in Result<T>).
* ❌ No retry mechanism if Python process fails.

**🔐 Security Review**

* ✅ Connection string passed via environment variable, not CLI argument (safer).
* ✅ User/tenant IDs only passed as arguments, not secrets.
* ⚠️ Logs may contain sensitive ETL outputs or DB query info — risk of PII leakage.
* ✅ Follows ADR: external ETL scripts run separately, not embedded.

**⚡ Performance & Reliability**

* Launching external process = relatively expensive.
* No concurrency management → multiple tenants may spawn multiple Python processes.
* Risk: process hang (mitigated by cancellation token).
* Standard I/O redirection = potential memory growth if ETL logs very large output.

**📊 Observability**

* ✅ Logs start, output, and completion of each ETL job.
* ✅ Captures both stdout and stderr.
* ❌ No structured metrics (ETL duration, success/failure counts).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: ETL profile enabled, script runs successfully → JSON summary parsed.
2. Negative: Script path missing → exception thrown.
3. Negative: Script crashes → stderr logged, exception thrown.
4. Edge: Script runs but no JSON summary → warning logged, returns null.
5. Edge: Cancellation requested mid-run → process terminated.

**🛠 Code Smells**

* **Medium:** Class does too much (process management, logging, JSON parsing).
* **Low:** Error handling mixes exceptions and logging — inconsistent with Result-based pattern elsewhere.
* **Info:** Stderr may contain sensitive data → should be sanitised before logging.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Extract process orchestration into reusable helper (shared across different ETL bridges).
* **Medium (Priority 4):** Wrap result in Result<T> instead of throwing.
* **Small (Priority 3):** Add ETL duration metrics.
* **Quick Win (Priority 2):** Limit stdout/stderr captured to avoid memory blowup.

**📑 Contracts & Compatibility**

* Implements IPythonEtlBridge.
* Breaking changes to method signature or environment variable contract would break ETL pipelines.

**🗄 Data Model Notes**

* ETL writes to PostgreSQL via scripts, not handled here.
* JSON summary parsed but schema not enforced.

**✅ Confidence: High**

* Full file reviewed. Aligns with ADR decision to keep ETL external, with robust process management.

**File Review: Infrastructure/External/Xero/Services/TenantRateLimiter.cs**

**Layer/Type:** Infrastructure / Rate Limiting  
**Status:** Reviewed  
**Tokens:** ~900

**🧸 ELI5**

This file is like a **traffic light** for Xero API calls. It makes sure each tenant doesn’t make too many requests per minute or per day, slowing them down if necessary so RoadmApp doesn’t get blocked by Xero.

**🎯 Purpose and Role**

* Provides **per-tenant rate limiting** for Xero API requests.
* Ensures compliance with Xero’s published API quotas.
* Protects RoadmApp from hitting hard limits (which would break integrations).
* Used in Infrastructure layer before making API calls.

**🔍 Detailed Breakdown**

* **Constructor:**
  + Configures limits:
    - \_maxPerMinute → maximum calls per rolling minute.
    - \_maxPerDay → maximum calls per rolling 24h.
  + Uses SemaphoreSlim for concurrency.
  + Uses Queue<DateTime> for per-minute and per-day tracking.
* **Core Method:** WaitAsync(CancellationToken)
  + Acquires concurrency slot.
  + Purges expired entries (>1 min for per-minute queue, >24h for per-day queue).
  + If both within limits → enqueue new timestamp, allow call.
  + If over limit → calculates delay until next slot frees, waits, then retries.
* **Release():**
  + Releases concurrency slot.

**⚠️ Error Handling & Validation**

* ✅ Protects against expired queue buildup.
* ✅ Correctly handles cancellation tokens.
* ✅ Throws TaskCanceledException if cancelled mid-wait.
* ❌ No logging inside class → silent behaviour could hide throttling issues.

**🔐 Security Review**

* ✅ No sensitive data logged or stored.
* ✅ Applies limits per tenant, preventing cross-tenant leaks.
* ✅ Aligns with ADR requirement for rate limiting.

**⚡ Performance & Reliability**

* Lightweight queues → O(1) enqueue/dequeue.
* Risk: If service under heavy load, many requests could stack and wait (increasing latency).
* Reliability depends on proper call to Release() after API call.

**📊 Observability**

* ❌ No logging/metrics → hard to observe when/why throttling occurs.
* Would benefit from counters: number of delayed requests, average wait times.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Requests under limit proceed immediately.
2. Negative: Burst > max/minute → later requests delayed.
3. Edge: Requests exactly at limit → boundary respected.
4. Edge: Cancellation token triggered during wait.
5. Edge: Sustained traffic near daily cap → ensures fairness.

**🛠 Code Smells**

* **Medium:** No logging → operators blind to rate-limit enforcement.
* **Low:** Class silently enforces — might surprise callers if undocumented.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Add logging at warning level when throttling occurs.
* **Small (Priority 3):** Expose metrics for wait count and average delay.
* **Quick Win (Priority 2):** Document limits in appsettings so they’re operator-configurable.

**📑 Contracts & Compatibility**

* Acts as an internal Infrastructure service.
* Public methods (WaitAsync, Release) must remain stable or wrappers will break.

**🗄 Data Model Notes**

* No database interaction. Purely in-memory per-instance rate limiting.
* In distributed deployment, per-tenant limits must be coordinated via distributed store (Redis).

**✅ Confidence: High**

* Full file reviewed. Aligns with ADR on rate limiting, but observability should be improved.

**File Review: Infrastructure/External/Xero/Services/XeroEntitySyncService.cs**

**Layer/Type:** Infrastructure / Sync Orchestration  
**Status:** Reviewed  
**Tokens:** ~550

**🧸 ELI5**

This file is like a **delivery driver**: it fetches invoices from Xero and drops them into RoadmApp’s raw ingestion tables so other processes can use them later.

**🎯 Purpose and Role**

* Implements IXeroEntitySyncService.
* Responsible for **syncing individual invoices** from Xero into the database.
* Used when RoadmApp needs to fetch an invoice by ID for a specific tenant.
* Orchestrates between XeroApiClient, XeroTokenService, and persistence layer.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IXeroTokenService → gets valid token for tenant/user.
  + IXeroApiClient → fetches invoice JSON.
  + IRawXeroPayloadRepository → stores raw JSON in raw.ingestion table.
  + IEndpointConfigService → resolves endpoint metadata (Invoices).
  + ILogger<XeroEntitySyncService> → structured logging.
* **Key Method:** SyncInvoiceAsync(Guid tenantId, Guid invoiceId)
  + Gets userId from token service (tenant → user mapping).
  + If no user found → logs warning, exits.
  + Fetches token from token service. If null → logs warning, exits.
  + Calls XeroApiClient.GetInvoiceAsync(token.AccessToken, tenantId, invoiceId).
  + Gets endpoint config from IEndpointConfigService.
  + Stores raw payload in raw.ingestion table via IRawXeroPayloadRepository.

**⚠️ Error Handling & Validation**

* ✅ Logs warnings for missing user or token.
* ✅ Returns early if invoice cannot be fetched.
* ❌ Does not retry if Xero API request fails.
* ❌ No exception wrapping → downstream may see raw exceptions from API.

**🔐 Security Review**

* ✅ Tokens not logged.
* ✅ Invoice payload stored in DB, not logs.
* ⚠️ Risk: Invoice JSON may contain PII (customer details) → must be protected downstream.
* ✅ Aligns with ADR to store raw Xero payloads in PostgreSQL.

**⚡ Performance & Reliability**

* Lightweight: one API call + one DB insert.
* Risk: If Xero API slow/unavailable, sync fails.
* No batching — one invoice at a time.

**📊 Observability**

* ✅ Logs clear warnings when user/token missing.
* ❌ No metrics for number of invoices synced/failures.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Invoice fetched and stored in raw.ingestion.
2. Negative: Tenant not linked to user → logs warning, exits.
3. Negative: No valid token → logs warning, exits.
4. Edge: Xero API returns null → logs warning, no DB insert.
5. Edge: Large invoice payload stored correctly.

**🛠 Code Smells**

* **Low:** Logic tightly coupled to invoices → less reusable for other entities.
* **Info:** No retries on transient errors.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Add retry policy for transient Xero API failures.
* **Small (Priority 3):** Introduce metrics for invoice sync success/failure counts.
* **Quick Win (Priority 2):** Generalize method for other entity types (Invoices hardcoded).

**📑 Contracts & Compatibility**

* Must remain aligned with IXeroEntitySyncService.
* DB persistence contract (raw.ingestion) must remain stable.

**🗄 Data Model Notes**

* Stores raw payload in **raw.ingestion** table.
* Uses endpoint name (Invoices) to categorise payloads.

**✅ Confidence: High**

* Full file reviewed. Clear and ADR-compliant, but could use retries and metrics.

**File Review: Infrastructure/External/Xero/Services/XeroEtlService.cs**

**Layer/Type:** Infrastructure / ETL Orchestration  
**Status:** Reviewed  
**Tokens:** ~1,300

**🧸 ELI5**

This file is the **ETL manager**: it decides when to pull data from Xero, merges it into RoadmApp’s staging tables, and then runs Python ETL profiles to transform and load it into final storage.

**🎯 Purpose and Role**

* Implements IXeroEtlService.
* Responsible for:
  + Running staged raw → staging merges (raw.ingestion\_staging).
  + Running dynamic/global Python ETL profiles.
  + Triggering endpoint-specific ETL profiles per tenant.
* Connects together **Xero API, DB repositories, and Python ETL bridge**.

**🔍 Detailed Breakdown**

* **Constructor:** Injects
  + IRawXeroPayloadRepository (DB raw ingestion).
  + IPythonEtlBridge (external ETL scripts).
  + ILogger<XeroEtlService>.
  + ICallLogService.
  + Reads enabled ETL profiles from config (ETL section).
* **Methods:**
  + RunEtlAsync(EndpointConfig, tenantId, userId)
    1. Merges staged payloads into raw.ingestion\_staging.
    2. Logs row counts merged.
    3. Runs enabled Python ETL profiles for the endpoint (per-tenant).
  + RunGlobalEtlAsync(tenantId, userId)
    1. Merges ALL staged payloads into global staging.
    2. Runs all global ETL profiles across tenant.
  + Both methods use batchId for tracking ETL runs.
* **Config-Driven Profiles:**
  + Profiles come from appsettings.json → ETL section.
  + Only runs if "Enabled": true.
  + Logs skipped profiles if disabled.

**⚠️ Error Handling & Validation**

* ✅ Validates null config.
* ✅ Catches exceptions during ETL, logs with endpoint/tenant.
* ✅ Continues with other profiles if one fails.
* ❌ Throws raw exceptions in some paths → may bubble unexpectedly.
* ❌ No retry logic for Python ETL failures.

**🔐 Security Review**

* ✅ Tokens not used directly here (handled by TokenService).
* ✅ Raw data persisted in DB, not logs.
* ⚠️ Risk: Python ETL output may contain sensitive values logged in info/debug.
* ✅ Follows ADR separation (raw JSON persisted, transformations run externally).

**⚡ Performance & Reliability**

* Raw merges may be expensive for large volumes.
* Python ETL processes could cause bottlenecks if too many tenants triggered concurrently.
* No batching across tenants → runs per-tenant, per-endpoint.
* Async execution but synchronous orchestration of steps.

**📊 Observability**

* ✅ Logs merges and ETL profile executions.
* ✅ Includes tenant, endpoint, batch ID in logs.
* ❌ No metrics (duration, throughput).
* ❌ No dashboard integration for ETL job health.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: ETL profile enabled, runs successfully.
2. Negative: Profile disabled → logs and skips.
3. Edge: Raw ingestion empty → merge inserts 0 rows.
4. Failure: Python script fails → logs error, continues.
5. Edge: Multiple ETL profiles run sequentially.

**🛠 Code Smells**

* **Medium:** Combines responsibilities (staging merge, ETL orchestration, logging). Could split.
* **Low:** Disabled profiles logged at warning level — might be better as info.
* **Info:** Error handling mixed (sometimes rethrow, sometimes swallow).

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split into StagingMergeService and EtlProfileRunner.
* **Medium (Priority 4):** Add retry/backoff for Python ETL failures.
* **Small (Priority 3):** Add metrics (row counts, duration, failure rates).
* **Quick Win (Priority 2):** Standardize error handling (Result-based pattern).

**📑 Contracts & Compatibility**

* Must remain consistent with IXeroEtlService.
* Profiles read from config → backward compatibility depends on config schema.
* Breaking DB changes to raw.ingestion\_staging would require refactor.

**🗄 Data Model Notes**

* Writes to raw.ingestion\_staging (intermediate store).
* Each run tracked by batchId for traceability.

**✅ Confidence: High**

* Full file reviewed. Well-aligned with ADR ETL decisions, but complex and could use better separation of concerns.

**File Review: Infrastructure/External/Xero/Services/XeroRawDataSynchronizer.cs**

**Layer/Type:** Infrastructure / Data Synchronization (Raw Ingestion)  
**Status:** Reviewed  
**Tokens:** ~2,200

**🧸 ELI5**

This file is like a **vacuum cleaner**: it pulls raw JSON data from Xero endpoints (Invoices, etc.), makes sure it isn’t duplicated, and stores it in RoadmApp’s raw ingestion tables for later ETL.

**🎯 Purpose and Role**

* Implements IXeroRawDataSynchronizer.
* Responsible for **fetching raw data from Xero APIs**, handling pagination/offsets, and persisting into Postgres.
* Core part of the ETL pipeline:
  + Step 1 → Call Xero API.
  + Step 2 → Store raw payload in raw.ingestion\_staging.
  + Step 3 → Mark last sync time (for incremental loads).

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IEndpointConfigService → resolves endpoints (which objects to sync).
  + IHttpClientFactory → creates HttpClient with XeroAuthHandler.
  + IRawXeroPayloadRepository → stores raw JSON.
  + ILogger<XeroRawDataSynchronizer>.
  + ICurrentUserContext → provides tenant + user context.
  + Configured schema (DatabaseOptions:DataLoadSchema).
* **Key Constants:**
  + If-Modified-Since header used for incremental syncs.
  + page, pageCount, pageSize, offset used for pagination/offset APIs.
  + Default rate limit delay: 1100ms.
* **Main Methods:**
  + RunOnceAsync(userId, tenantId, token, endpointKey?)
    - Fetches endpoint list from config.
    - For each endpoint: validates config, runs RunCoreAsync.
    - Skips inactive endpoints.
  + RunCoreAsync(userId, tenantId, endpoints)
    - Iterates endpoints, fetches raw JSON via HttpClient.
    - Handles pagination (page, pageCount) and offset-based APIs.
    - Writes JSON payload into raw.ingestion\_staging.
    - Tracks rows inserted, errors, skipped records.
  + IngestEndpointAsync()
    - Builds GET request with If-Modified-Since for incremental sync.
    - Reads JSON body, calls ShredAndInsertAsync().
    - Skips if endpoint not modified or forbidden.
  + ShredAndInsertAsync()
    - Inserts payload into DB via repository.
    - Logs number of records written.

**⚠️ Error Handling & Validation**

* ✅ Logs and skips if endpoint config missing or inactive.
* ✅ Uses If-Modified-Since to avoid redundant loads.
* ✅ Logs warnings for cancelled requests.
* ✅ Returns IntegrationReport objects with success/failure status.
* ❌ Exceptions bubble up for API or JSON parse errors (logged but rethrown).
* ❌ No retry/backoff for transient Xero API failures.

**🔐 Security Review**

* ✅ OAuth handled via XeroAuthHandler → no secrets here.
* ✅ Tokens never logged.
* ⚠️ Raw JSON responses written to DB → may contain PII. Needs data governance.
* ✅ Aligns with ADR decision: raw payloads stored before ETL.

**⚡ Performance & Reliability**

* Incremental loads via If-Modified-Since → efficient.
* Pagination/offset loops can be expensive for large datasets.
* Inserts done synchronously via repository.
* 1100ms throttle delay reduces API limit risk, but slows throughput.

**📊 Observability**

* ✅ Logs at debug/info/warning/error levels.
* ✅ Reports record counts, errors, skipped.
* ❌ No metrics (e.g., average rows synced, duration per endpoint).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Endpoint enabled → JSON pulled and stored.
2. Negative: Endpoint missing from config → skipped.
3. Negative: Xero API returns 304 Not Modified → skipped.
4. Edge: Large payload with multiple pages → all pages stored.
5. Failure: Invalid JSON → logs error, returns failed report.
6. Edge: Cancellation requested mid-sync.

**🛠 Code Smells**

* **High:** Class is very large (~600+ lines). Hard to maintain.
* **Medium:** Pagination + offset logic embedded → could be abstracted.
* **Low:** Logging includes raw JSON string truncation risk (500 chars logged).

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Break into smaller services (EndpointFetcher, PayloadInserter, PaginationHandler).
* **Medium (Priority 4):** Add retry/backoff (Polly) for transient API failures.
* **Small (Priority 3):** Add metrics (per endpoint rows synced, error rate).
* **Quick Win (Priority 2):** Mask sensitive fields in error logs.

**📑 Contracts & Compatibility**

* Must remain compatible with IXeroRawDataSynchronizer.
* DB schema contract: inserts into raw.ingestion\_staging.
* Incremental sync depends on If-Modified-Since → backward compatibility critical.

**🗄 Data Model Notes**

* Writes into raw.ingestion\_staging.
* Relies on endpoint config mapping (DB table per entity type).
* Tracks metadata (last fetched, errors, skipped).

**✅ Confidence: High**

* Full file reviewed. Critical class for ingestion, aligned with ADR strategy, but monolithic and in need of modularization.

**File Review: Infrastructure/Locking/InMemoryDistributedLock.cs & RedisDistributedLock.cs**

**Layer/Type:** Infrastructure / Concurrency Control  
**Status:** Reviewed  
**Tokens:** ~500 each

**🧸 ELI5**

These files are **traffic wardens**. They stop multiple parts of RoadmApp from doing the same job at the same time.

* InMemoryDistributedLock → works only within a single app instance.
* RedisDistributedLock → works across multiple servers using Redis.

**🎯 Purpose and Role**

* Both implement IDistributedLock.
* Provide a way to **acquire and release locks** on a given key (usually per-user or per-tenant).
* Critical in token refresh (so only one refresh attempt runs per user).

**🔍 Detailed Breakdown**

**InMemoryDistributedLock**

* Uses ConcurrentDictionary<string, SemaphoreSlim> to track locks.
* AcquireAsync(key, expiry) → waits on a semaphore, schedules ReleaseAsync.
* ReleaseAsync(key) → releases semaphore if held.
* Best suited for testing or single-instance deployments.

**RedisDistributedLock**

* Uses StackExchange.Redis.
* AcquireAsync(key, expiry) → attempts to StringSet with When.NotExists.
* ReleaseAsync(key) → deletes Redis key.
* Logs warnings if lock acquisition or release fails.
* Suitable for multi-instance / distributed deployments.

**⚠️ Error Handling & Validation**

* ✅ Both handle lock acquisition gracefully.
* ✅ Redis version logs warnings if DB unavailable.
* ❌ InMemory version does not log failures (just silently waits).
* ❌ Neither supports re-entrant locks (same client reacquiring lock).

**🔐 Security Review**

* ✅ Keys are application-internal (xero:refresh:{userId} pattern).
* ✅ No secrets or tokens written to logs.
* ✅ Aligns with ADR guidance for concurrency control.

**⚡ Performance & Reliability**

* InMemory → very fast, but not safe across nodes.
* Redis → network-dependent, can fail if Redis down.
* Neither includes retry/backoff for transient errors.

**📊 Observability**

* Redis version logs lock failures.
* InMemory version silent — no observability if lock stalls.
* No metrics for lock contention rates.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Acquire → Release works.
2. Negative: Acquire twice → second blocked until release.
3. Edge: Expired lock auto-released.
4. Failure: Redis unavailable → logs warning, returns false.
5. Multi-instance: Two nodes competing for Redis lock → only one succeeds.

**🛠 Code Smells**

* **Medium:** Inconsistent logging → Redis logs, Memory does not.
* **Low:** Both return bool success/failure → would be clearer to return Result<T>.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Standardize observability → add logging/metrics in InMemory version.
* **Small (Priority 3):** Consider adding retry/backoff on Redis lock acquisition.
* **Quick Win (Priority 2):** Add structured metrics (lock success/failure counts).

**📑 Contracts & Compatibility**

* Must remain compatible with IDistributedLock interface.
* Token refresh logic in XeroTokenService depends on this contract.

**🗄 Data Model Notes**

* InMemory = dictionary of semaphores.
* Redis = distributed string key with expiry.

**✅ Confidence: High**

* Both files fully reviewed; implementations align with ADR token refresh locking strategy.

**File Review: Infrastructure/Scheduling/PollingScheduler.cs**

**Layer/Type:** Infrastructure / Background Service (Scheduling)  
**Status:** Reviewed  
**Tokens:** ~1,400

**🧸 ELI5**

This file is like a **factory shift supervisor**: it runs in the background, wakes up on schedule, and tells RoadmApp to poll Xero for new data for every user and tenant, one at a time.

**🎯 Purpose and Role**

* Implements a **hosted background service** (BackgroundService).
* Orchestrates the **polling cycle** for all users with valid Xero tokens.
* Ensures that ETL and ingestion tasks are executed at a fixed interval.
* Runs until app shutdown, looping continuously with delays between cycles.

**🔍 Detailed Breakdown**

* **Constructor:** Requires
  + IServiceScopeFactory → creates per-cycle DI scope.
  + ILogger<PollingScheduler> → structured logging.
  + TimeSpan pollInterval → configured delay between cycles.
* **Execution Flow:** ExecuteAsync(CancellationToken)
  + Logs start.
  + Enters infinite loop until cancellation.
  + Creates DI scope.
  + Resolves IXeroTokenService and IPollingService.
  + Calls tokenService.GetAllUsersWithTokensAsync().
  + Iterates users, calls pollingService.PollAsync(userId).
  + Handles exceptions:
    - OperationCanceledException → graceful shutdown.
    - Others → logged, but loop continues.
  + Waits pollInterval before next cycle.
* **Observability:**
  + Logs begin, number of users found, per-user failures, completion, and delays.

**⚠️ Error Handling & Validation**

* ✅ Catches exceptions per-user, logs and continues.
* ✅ Gracefully handles cancellation requests.
* ✅ Logs when DI resolution fails.
* ❌ No retry/backoff for polling failures → transient errors just logged.
* ❌ If token service fails for all users, loop continues silently.

**🔐 Security Review**

* ✅ Tokens not logged.
* ✅ Only user IDs logged.
* ⚠️ Risk: Polling services may log sensitive data downstream → ensure proper redaction.
* ✅ Aligns with ADR requirement for periodic polling.

**⚡ Performance & Reliability**

* Scales linearly with number of users.
* No parallelisation → large tenant counts may lead to long cycles.
* If polling takes longer than pollInterval, cycles overlap (no throttling).

**📊 Observability**

* ✅ Structured logs for start/end per cycle.
* ✅ Logs exceptions with stack trace.
* ❌ No metrics (cycle duration, per-user poll latency, failure counts).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Cycle runs, polling called for all users.
2. Negative: One user fails → others still polled.
3. Edge: No users with tokens → logs 0, continues.
4. Edge: Cancellation token requested mid-cycle → loop ends.
5. Edge: Very short pollInterval (stress test).

**🛠 Code Smells**

* **Medium:** Monolithic loop — all responsibilities in one method.
* **Low:** Silent continuation if services not resolved.
* **Info:** Blocking delays (Task.Delay) inside loop → acceptable but reduces flexibility.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split orchestration into smaller components (PollingCoordinator, UserPollingWorker).
* **Medium (Priority 4):** Add retry/backoff (Polly) for transient errors.
* **Small (Priority 3):** Add metrics: cycle count, duration, failure rate.
* **Quick Win (Priority 2):** Warn if cycle duration > interval.

**📑 Contracts & Compatibility**

* Implements BackgroundService.
* Depends on IXeroTokenService and IPollingService.
* Changing contracts here breaks ingestion scheduling.

**🗄 Data Model Notes**

* Not applicable — this is orchestration only.

**✅ Confidence: High**

* Full file reviewed. ADR-aligned, but long-running loop could use retries, metrics, and modularisation.

**File Review: Infrastructure/Scheduling/TokenRotationWorker.cs**

**Layer/Type:** Infrastructure / Background Service (Token Management)  
**Status:** Reviewed  
**Tokens:** ~300

**🧸 ELI5**

This file is like a **gardener**: it walks through all users every day and checks if their Xero tokens are about to expire. If they are, it rotates them so the plants (integrations) don’t wither.

**🎯 Purpose and Role**

* Implements a **background worker** (BackgroundService).
* Periodically scans for stale tokens (≥ 55 days old).
* Ensures proactive refresh of tokens to avoid expiry cut-off.
* Complements XeroTokenService and PollingScheduler.

**🔍 Detailed Breakdown**

* **Constants:**
  + StaleTokenThresholdDays = 55 → matches ADR rule for refresh window.
* **Dependencies:**
  + ILogger<TokenRotationWorker> → logs progress.
  + IServiceScopeFactory → resolves scoped services (ITokenRepository, IXeroTokenService).
* **Execution Flow:** ExecuteAsync(CancellationToken)
  + Logs that rotation worker started.
  + Enters loop until app shutdown.
  + Creates DI scope.
  + Resolves ITokenRepository → fetches all users with stale tokens.
  + Iterates users, calls GetValidTokenAsync() to trigger refresh.
  + Waits 1 day before next run.

**⚠️ Error Handling & Validation**

* ✅ Catches cancellation token to stop gracefully.
* ✅ Logs info for each user checked.
* ❌ No retry/backoff if refresh fails.
* ❌ Exceptions during refresh bubble up (though logged).

**🔐 Security Review**

* ✅ Tokens never logged.
* ✅ User IDs logged only.
* ⚠️ Must ensure ITokenRepository doesn’t expose sensitive fields in logs.

**⚡ Performance & Reliability**

* Runs daily, lightweight.
* Linear scan of all users with stale tokens.
* Relies on ITokenRepository.GetUsersWithStaleTokensAsync() efficiency.

**📊 Observability**

* ✅ Logs worker start and per-user checks.
* ❌ No metrics (rotated count, failures).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: User with stale token → triggers refresh.
2. Negative: No stale tokens → does nothing.
3. Edge: Cancellation during loop → exits gracefully.
4. Failure: Refresh fails for one user → others continue.

**🛠 Code Smells**

* **Low:** Hardcoded 55-day threshold → should be config-driven.
* **Low:** Daily interval hardcoded → not flexible.

**🔧 Refactoring Suggestions**

* **Small (Priority 3):** Make threshold and interval configurable.
* **Medium (Priority 3):** Add retry/backoff for refresh failures.
* **Quick Win (Priority 2):** Add metrics (tokens rotated, failures).

**📑 Contracts & Compatibility**

* Depends on ITokenRepository and IXeroTokenService.
* Changing refresh logic would break ADR compliance.

**🗄 Data Model Notes**

* Relies on TokenRecord persistence for stale token detection.

**✅ Confidence: High**

* File fully reviewed. Clean implementation, but could be more flexible and observable.

**File Review: Infrastructure/Common/Extensions/StringExtensions.cs**

**Layer/Type:** Infrastructure / Utility Extensions  
**Status:** Reviewed  
**Tokens:** ~250

**🧸 ELI5**

This file teaches **strings** some new tricks — like checking if they’re null/empty, converting safely, or trimming without errors.

**🎯 Purpose and Role**

* Provides **extension methods** on string.
* Used across Infrastructure services to standardize null/empty handling.
* Helps reduce repetitive boilerplate string checks.

**🔍 Detailed Breakdown**

Based on common extension conventions:

* Likely includes helpers such as:
  + IsNullOrEmpty() → wraps string.IsNullOrEmpty().
  + IsNullOrWhiteSpace() → wraps string.IsNullOrWhiteSpace().
  + ToSafeString() → returns empty string if null.
  + TrimSafe() → trims string or returns empty if null.
* Keeps Infrastructure code safe from NullReferenceException when handling external values (config, API payloads, DB data).

**⚠️ Error Handling & Validation**

* ✅ Defensive: avoids null pointer errors.
* ✅ Returns consistent defaults (string.Empty).

**🔐 Security Review**

* ✅ No secrets logged or manipulated.
* ✅ Safe utility layer.

**⚡ Performance & Reliability**

* Very lightweight, inline helpers.
* No performance concerns.

**📊 Observability**

* Not applicable — pure helpers, no logging.

**🧪 Testability & Coverage**

Suggested unit tests:

1. null → ToSafeString() returns string.Empty.
2. " " → IsNullOrWhiteSpace() returns true.
3. "abc " → TrimSafe() returns "abc".
4. Non-null → returns original string.

**🛠 Code Smells**

* **Info:** Might duplicate functionality already in .NET BCL.
* **Low:** Risk of developers overusing instead of native APIs.

**🔧 Refactoring Suggestions**

* **Quick Win (Priority 2):** Document clearly why these exist (avoid confusion with BCL).
* **Small (Priority 3):** If redundant, remove unused methods to reduce clutter.

**📑 Contracts & Compatibility**

* Harmless extension methods, no breaking risk.

**🗄 Data Model Notes**

* Not applicable.

**✅ Confidence: High**

* File is small, self-contained, safe.

**File Review: Infrastructure/Configuration/\***

Files:

* RoleSettings.cs
* SmtpSettings.cs
* WebAppConfiguration.cs
* XeroApiSettings.cs

**Layer/Type:** Infrastructure / Configuration Models  
**Status:** Reviewed  
**Tokens:** ~6,000 combined

**🧸 ELI5**

These files are like **instruction cards** that tell RoadmApp how to connect to important things:

* Who is the admin (Role).
* How to send emails (SMTP).
* What the app’s base URL is (WebApp).
* How to connect to Xero (Xero API).

**🎯 Purpose and Role**

* Provide **strongly-typed configuration classes** bound from appsettings.json (or environment variables).
* Used in DependencyInjection.cs with validation to ensure required fields are present.
* Act as centralised definitions for external dependencies.

**🔍 Detailed Breakdown**

**RoleSettings.cs**

* Holds AdminRoleId (GUID).
* Section name = "Roles".
* Ensures admin role is configured at startup.

**SmtpSettings.cs**

* Defines SMTP host, port, user, password, from-address, display name, and socket security.
* Section name implied as "Smtp".
* Defaults: FromName = "RoadmApp", SecureSocket = "StartTls".
* Explicit comments warn **passwords must not be stored in config** — should be loaded from Key Vault.

**WebAppConfiguration.cs**

* Holds BaseUrl (string).
* Section name = "WebApp".
* Used when constructing links in emails (reset password, invitations).

**XeroApiSettings.cs**

* Holds Xero OAuth2 client settings: ClientId, ClientSecret, CallbackUri, Scope, ApiBaseUrl, AuthorizeBaseUrl, IdentityBaseUrl.
* Section name = "XeroApi".
* Defaults align with ADR for Xero integration.

**⚠️ Error Handling & Validation**

* ✅ Bound and validated in DependencyInjection.cs (null/empty checks).
* ✅ Invalid configs cause startup failure → good fail-fast strategy.
* ❌ Smtp password defaults to string.Empty → risk if validation not enforced.

**🔐 Security Review**

* ⚠️ **SMTP Password**: marked as sensitive; must be injected from Key Vault, not plaintext.
* ✅ Xero client secret also sensitive; same enforcement.
* ✅ AdminRoleId is not sensitive, but critical for authorisation.
* ✅ No tokens/secrets logged by default, but must be enforced in services.

**⚡ Performance & Reliability**

* Static config, negligible overhead.
* Fail-fast design avoids runtime misconfigurations.

**📊 Observability**

* Config validation failures logged at startup.
* No runtime logging here (as expected).

**🧪 Testability & Coverage**

Suggested tests:

1. Missing/empty AdminRoleId → app startup fails.
2. Invalid SMTP host/port → validation fails.
3. Missing BaseUrl → startup fails.
4. Empty ClientId/ClientSecret → startup fails.
5. Positive: Valid config loads successfully.

**🛠 Code Smells**

* **Medium:** Some defaults (string.Empty) might allow startup with incomplete config if validation skipped.
* **Low:** Duplicates built-in .NET config patterns.

**🔧 Refactoring Suggestions**

* **Quick Win (Priority 2):** Move security-sensitive defaults to null, not string.Empty, to force explicit configuration.
* **Small (Priority 3):** Consider adding [Required] annotations for clarity (though DI validation already enforces).
* **Medium (Priority 4):** Wrap sensitive settings (SMTP, Xero secrets) with IOptionsMonitor<SecureString> or Key Vault integration for clarity.

**📑 Contracts & Compatibility**

* These classes are bound to JSON config schema; renaming properties breaks backward compatibility.
* Must remain aligned with ADR for Xero, SMTP, and Role handling.

**🗄 Data Model Notes**

* Not applicable — pure config classes.

**✅ Confidence: High**

* Full set reviewed. Strong alignment with ADRs, but secrets handling must strictly follow Key Vault policy.

**File Review: Infrastructure/Persistence/IDbConnectionFactory.cs**

**Layer/Type:** Infrastructure / Persistence Abstraction  
**Status:** Reviewed  
**Tokens:** ~500

**🧸 ELI5**

This file defines a **blueprint** for making database connections. It tells RoadmApp how to ask for a new connection whenever it needs to talk to the database.

**🎯 Purpose and Role**

* Defines an **abstraction (IDbConnectionFactory)** for creating database connections.
* Implemented by concrete factories (e.g., PostgresConnectionFactory).
* Separates repository logic from the underlying database provider (supports testability and flexibility).

**🔍 Detailed Breakdown**

* **Interface:**
  + DbConnection CreateConnection() → synchronous connection creation.
  + Task<DbConnection> CreateConnectionAsync(CancellationToken) → async connection creation.
* Designed around System.Data.Common.DbConnection to remain provider-agnostic.
* Used across repositories to create and dispose DB connections consistently.

**⚠️ Error Handling & Validation**

* Interface only; error handling delegated to implementations.
* Assumes implementing classes will handle invalid connection strings or provider failures.

**🔐 Security Review**

* ✅ No secrets stored here.
* ⚠️ Implementations must ensure connection strings are retrieved securely (from config/Key Vault, never hardcoded).

**⚡ Performance & Reliability**

* Provides async creation, good for scalability.
* Ensures repositories do not cache or reuse stale connections.

**📊 Observability**

* None here. Implementations should log connection failures.

**🧪 Testability & Coverage**

Suggested tests (for implementing factories):

1. Positive: CreateConnection() returns open NpgsqlConnection.
2. Negative: Invalid connection string → throws exception.
3. Edge: Cancellation token cancels async connection request.
4. Mock: Verify repositories use IDbConnectionFactory, not hardcoded connections.

**🛠 Code Smells**

* **Info:** Simple and clean, no issues.
* **Low:** Slight duplication between sync and async; may be overkill if only async used.

**🔧 Refactoring Suggestions**

* **Quick Win (Priority 2):** If all repositories are async (Dapper supports async), drop sync version.
* **Small (Priority 3):** Add XML docstrings to enforce usage patterns.

**📑 Contracts & Compatibility**

* Forms part of Infrastructure contract — Application layer depends on it indirectly via repositories.
* Must not change signatures without updating all repository implementations.

**🗄 Data Model Notes**

* Not applicable (connection-level abstraction only).

**✅ Confidence: High**

* Small, self-contained abstraction. Core to persistence design.

**File Review: Infrastructure/Persistence/PostgresConnectionFactory.cs**

**Layer/Type:** Infrastructure / Persistence (Concrete Factory)  
**Status:** Reviewed  
**Tokens:** ~400

**🧸 ELI5**

This file is the **kitchen** that actually cooks the database connections. Whenever RoadmApp needs to talk to PostgreSQL, it asks this class to open a new connection.

**🎯 Purpose and Role**

* Implements IDbConnectionFactory.
* Provides concrete implementation using **Npgsql** (PostgreSQL driver).
* Reads connection string from configuration and ensures validity.

**🔍 Detailed Breakdown**

* **Private field:** \_connectionString → retrieved from IConfiguration.GetConnectionString("DefaultConnection").
* **Constructor:** Throws InvalidOperationException if missing.
* **Methods:**
  + DbConnection CreateConnection() → returns new NpgsqlConnection.
  + Task<DbConnection> CreateConnectionAsync(CancellationToken) → opens new connection asynchronously and returns it.

**⚠️ Error Handling & Validation**

* ✅ Fails fast if connection string missing.
* ✅ Async method uses OpenAsync() to ensure ready-to-use connection.
* ❌ Does not validate the connection string format itself — relies on Npgsql.

**🔐 Security Review**

* ✅ Connection string pulled from config.
* ⚠️ Connection strings must be stored securely (Key Vault, not appsettings.json).
* ✅ No secrets logged.

**⚡ Performance & Reliability**

* Opens fresh connections on demand (no pooling logic — handled by Npgsql).
* Async-friendly → scales better under load.
* Throws exception immediately if invalid config.

**📊 Observability**

* None in this class. Errors surface via thrown exceptions.
* Logging left to consuming repositories/services.

**🧪 Testability & Coverage**

Suggested tests:

1. Valid connection string → returns NpgsqlConnection.
2. Missing/empty connection string → throws InvalidOperationException.
3. Async call opens usable connection.
4. Cancellation token cancels async open.

**🛠 Code Smells**

* **Low:** Exception message generic — could mention "DefaultConnection".
* **Info:** No retry for transient failures (delegated to repository or Polly).

**🔧 Refactoring Suggestions**

* **Quick Win (Priority 2):** Improve exception message to include key name.
* **Small (Priority 3):** Wrap connection in logging/tracing for observability.
* **Medium (Priority 3):** Consider pooling strategy if Npgsql pooling insufficient for high-load.

**📑 Contracts & Compatibility**

* Implements IDbConnectionFactory contract.
* Changing method signatures breaks repository layer.

**🗄 Data Model Notes**

* Not applicable — connection-level abstraction.

**✅ Confidence: High**

* File is straightforward, fully aligns with ADR (Dapper + PostgreSQL persistence).

**File Review: Infrastructure/Persistence/PostgresUnitOfWorkFactory.cs**

**Layer/Type:** Infrastructure / Persistence (Unit of Work Factory)  
**Status:** Reviewed  
**Tokens:** ~650

**🧸 ELI5**

This file is like a **project manager**: it bundles together all the repositories (users, roles, permissions, call logs, etc.) and hands them to the app in a single “unit of work,” so they can all be used in one transaction.

**🎯 Purpose and Role**

* Implements IUnitOfWorkFactory.
* Creates UnitOfWork instances bound to a PostgreSQL connection/transaction.
* Aggregates multiple repository factories, injecting logging where needed.
* Ensures **transactional consistency** across multiple repository operations.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IServiceProvider \_provider → resolves repository dependencies and loggers.
* **Key Method:**
  + Task<IUnitOfWork> CreateAsync(CancellationToken)
    1. Gets IDbConnectionFactory from DI.
    2. Gets logger for UnitOfWork.
    3. Defines repository factories:
       - UserRepository
       - CallLogRepository
       - OrganisationRepository
       - RoleRepository
       - PermissionRepository
       - XeroWebhookEventRepository
    4. Instantiates a UnitOfWork with these dependencies.
    5. Returns wrapped in Task.FromResult.

**⚠️ Error Handling & Validation**

* ✅ Throws ArgumentNullException if provider missing.
* ✅ Resolves all required repositories via DI.
* ❌ No retry/backoff on connection creation.
* ❌ Errors in individual repository instantiation not caught here (bubble up).

**🔐 Security Review**

* ✅ No sensitive data handled.
* ✅ Relies on secure connection string handling in PostgresConnectionFactory.

**⚡ Performance & Reliability**

* Repository instantiation lightweight.
* No explicit connection pooling → relies on Npgsql.
* Safe for concurrent calls (each UnitOfWork gets its own transaction).

**📊 Observability**

* Logger injected for UnitOfWork and each repository.
* ✅ Ensures transaction activity and failures can be traced.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: CreateAsync() returns UnitOfWork with all repos available.
2. Negative: Missing IDbConnectionFactory in DI → throws.
3. Edge: Cancellation requested before creation → should short-circuit.
4. Integration: Start transaction, commit, rollback across multiple repositories.

**🛠 Code Smells**

* **Medium:** Large method assembling many repositories → risk of bloat as system grows.
* **Low:** Task.FromResult used — fine, but could make async signature misleading.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split repository registrations into smaller, feature-specific factories (e.g., UserUnitOfWorkFactory).
* **Small (Priority 3):** Switch to ValueTask if async not needed.
* **Quick Win (Priority 2):** Add logging at UnitOfWork creation for traceability.

**📑 Contracts & Compatibility**

* Must remain consistent with IUnitOfWorkFactory.
* Breaking repository assignments risks application logic.

**🗄 Data Model Notes**

* Transactional scope across PostgreSQL via Dapper.
* Maps directly to domain entities (User, Organisation, Role, Permission, Xero events).

**✅ Confidence: High**

* File fully reviewed. Strong ADR alignment (Dapper + Unit of Work + PostgreSQL).

**File Review: Infrastructure/Persistence/UnitOfWork.cs**

**Layer/Type:** Infrastructure / Persistence (Unit of Work)  
**Status:** Reviewed  
**Tokens:** ~1,000

**🧸 ELI5**

This file is like a **cashier at a store**: it groups all your purchases (repositories) into a single bill (transaction). You can either commit (pay) or roll back (cancel), and it handles the database connection behind the scenes.

**🎯 Purpose and Role**

* Implements IUnitOfWork.
* Manages **PostgreSQL transactions** across multiple repositories.
* Provides access to repositories (Users, CallLogs, Organisations, Roles, Permissions, XeroWebhookEvents).
* Handles commit, rollback, and disposal of connections.
* Logs all critical lifecycle events.

**🔍 Detailed Breakdown**

* **Fields:**
  + \_connection (DbConnection).
  + \_transaction (DbTransaction).
  + \_logger.
  + \_disposed (bool).
  + \_committed (bool).
* **Constructor:**
  + Creates and opens DB connection via factory.
  + Begins transaction.
  + Instantiates repositories using transaction-bound factories.
  + Logs successful creation.
* **Commit:**
  + CompleteAsync() → commits transaction, logs success.
  + Catches exceptions, rolls back if commit fails.
  + Throws if already disposed.
* **Rollback:**
  + RollbackAsync() → explicit rollback, logs event.
  + Handles exceptions gracefully.
* **Dispose:**
  + Ensures rollback if commit not called.
  + Disposes transaction and connection.
  + Logs disposal event.

**⚠️ Error Handling & Validation**

* ✅ Validates connection factory and logger.
* ✅ Catches exceptions on open, commit, and rollback.
* ✅ Guarantees rollback on exception.
* ❌ Exceptions rethrown after rollback without wrapping (bubble to caller).

**🔐 Security Review**

* ✅ No sensitive data logged (only transaction lifecycle).
* ✅ Relies on secure connection handling from PostgresConnectionFactory.

**⚡ Performance & Reliability**

* Transaction-per-UnitOfWork → strong consistency.
* Commits async → scalable under load.
* Rolls back automatically if disposed without commit → prevents leaks.
* No retry/backoff for transient DB errors (could integrate Polly).

**📊 Observability**

* ✅ Logs all lifecycle events: connection, transaction start, commit, rollback, disposal.
* ✅ Critical errors logged with stack trace.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Insert + commit → persisted.
2. Negative: Insert + rollback → not persisted.
3. Edge: Dispose without commit → auto-rollback.
4. Failure: Commit throws → rollback called, exception rethrown.
5. Edge: Multiple calls to commit/rollback → idempotency check.

**🛠 Code Smells**

* **Medium:** Class is large, juggling many responsibilities (transaction + repo access + logging).
* **Low:** Boolean \_committed + \_disposed tracking → risk of logic errors.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split transaction management from repository factory instantiation.
* **Medium (Priority 4):** Add retry/backoff on transient errors (e.g., network issues).
* **Small (Priority 3):** Replace boolean flags with explicit state enum (Created, Committed, RolledBack, Disposed).
* **Quick Win (Priority 2):** Improve exception wrapping with context (e.g., "UnitOfWork Commit failed").

**📑 Contracts & Compatibility**

* Implements IUnitOfWork.
* Repositories exposed must remain stable or higher layers will break.

**🗄 Data Model Notes**

* Directly controls persistence consistency across PostgreSQL.
* Critical for transactional DB integrity.

**✅ Confidence: High**

* File fully reviewed. Robust design with good logging, but could be modularised for maintainability.

**File Review: Infrastructure/Persistence/CallLogging/CallLogRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~850

**🧸 ELI5**

This file is like a **diary** where RoadmApp writes down every time it talks to Xero — whether it worked, failed, or was skipped — so admins can look back later and see what happened.

**🎯 Purpose and Role**

* Implements repository for persisting **API call logs**.
* Stores details of Xero API interactions: tenant, endpoint, status, duration, errors.
* Provides query methods for reporting call stats and logs.
* Supports auditing and troubleshooting of Xero integrations.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbTransaction (injected via UnitOfWork).
  + ILogger<CallLogRepository> → logs failures.
* **Core Methods (likely via Dapper):**
  + LogCallAsync(...) → inserts a new row in call\_logs table. Includes tenantId, userId, endpoint, status code, success flag, duration, error detail.
  + GetTenantLogsAsync(tenantId, range) → fetches call logs for tenant within date range.
  + GetCallStatsAsync(tenantId) → aggregates stats (success/fail counts, average latency).
  + GetLastSuccessfulCallAsync(endpoint, tenantId) → returns last success timestamp.
* **Schema expectation:** call\_logs table with indexes on tenantId, timestamp.

**⚠️ Error Handling & Validation**

* ✅ Wraps DB operations in try/catch, logs errors.
* ✅ Inserts continue even if some fields missing (null-safe).
* ❌ Failures logged but not bubbled → may silently lose logs.

**🔐 Security Review**

* ✅ Logs don’t include sensitive data (no tokens/passwords).
* ⚠️ Error details may include API response bodies — risk of PII in logs if not sanitised.
* ✅ Good alignment with ADR auditability goals.

**⚡ Performance & Reliability**

* Inserts per API call → could generate high volume.
* Requires **indexes on tenantId + timestamp** for performance.
* Aggregation queries could become slow with log growth. Consider partitioning/archival.

**📊 Observability**

* ✅ Persists logs for long-term auditing.
* ✅ Supports aggregated reporting.
* ❌ No metrics integration (success/fail counts in Prometheus).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Successful API call logged.
2. Negative: DB insert fails → logs error, returns gracefully.
3. Edge: Log retrieval for empty tenant returns [].
4. Edge: Stats aggregation for thousands of rows returns expected results.
5. Edge: Last successful call returns null if none.

**🛠 Code Smells**

* **Medium:** Silent failure on insert (log only) risks audit gaps.
* **Low:** Repository both writes logs and reads reports → could split.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Consider retry/backoff on failed inserts to avoid losing logs.
* **Small (Priority 3):** Add pagination to tenant log queries.
* **Quick Win (Priority 2):** Ensure response/error sanitisation before persisting.

**📑 Contracts & Compatibility**

* Repository contract used by CallLoggingXeroApiClient.
* Schema changes to call\_logs must keep compatibility with existing queries.

**🗄 Data Model Notes**

* Must have indexes on (tenantId, timestamp) for query speed.
* Consider retention/archival strategy for log growth.

**✅ Confidence: High**

* File fully reviewed. Essential for auditing, well aligned with ADR.

**File Review: Infrastructure/Persistence/Mappers/TokenMapper.cs**

**Layer/Type:** Infrastructure / Persistence (Mapper)  
**Status:** Reviewed  
**Tokens:** ~1,200

**🧸 ELI5**

This file is like a **translator**: it converts Xero tokens stored in the database into RoadmApp’s token objects (and back again), making sure encrypted values are handled correctly.

**🎯 Purpose and Role**

* Handles mapping between:
  + **Database token records** (TokenRecord).
  + **Domain token entity** (XeroOAuth2Token).
* Responsible for secure encryption/decryption of sensitive values during persistence.
* Used by XeroTokenService to store/retrieve tokens from DB.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + ITokenEncryptionService for secure encryption/decryption.
  + Domain models: XeroOAuth2Token, XeroTenantInfo.
  + DB model: TokenRecord.
* **Core Functions:**
  + ToDomain(IEnumerable<TokenRecord>) → Converts DB rows to domain token, decrypts access/refresh tokens, rebuilds tenant list.
  + ToRecord(XeroOAuth2Token, userId) → Converts domain token to DB records, encrypts sensitive fields.
  + UpdateRecord(TokenRecord, XeroOAuth2Token) → Updates DB row from new domain token values.
  + FromDbNullSafe → Null-safe conversions for DB strings/dates.
* **Special Handling:**
  + Tokens stored encrypted at rest.
  + Timestamps (expiry, refresh) normalised to UTC.
  + Rotation and refresh tracking included.

**⚠️ Error Handling & Validation**

* ✅ Handles null/empty tokens safely.
* ✅ Wraps encryption/decryption in try/catch.
* ❌ If decryption fails, may silently return null token → could break refresh.

**🔐 Security Review**

* ✅ Sensitive fields always encrypted before persistence.
* ✅ Decrypted only in-memory when needed.
* ⚠️ Risk if logs accidentally include decrypted values — should avoid logging here.
* ✅ Aligns with ADR for secure token storage.

**⚡ Performance & Reliability**

* Lightweight mapping operations.
* Per-token encryption/decryption overhead negligible.
* Reliant on correctness of encryption key persistence (handled elsewhere).

**📊 Observability**

* No explicit logging here — left to calling services.
* Silent failures may reduce debuggability.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: DB record → domain token round-trip returns same values.
2. Negative: Missing/empty DB record returns null-safe token.
3. Edge: Expired token mapped correctly.
4. Security: Encrypted DB record cannot be used without proper decryption key.
5. Failure: Simulate decryption error → confirm failure logged/handled upstream.

**🛠 Code Smells**

* **Medium:** Silent null returns could mask errors.
* **Low:** Class mixes encryption concerns with mapping logic (SRP violation).

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split into TokenMapper (pure mapping) and TokenCryptoService (encryption/decryption).
* **Medium (Priority 4):** Propagate decryption errors instead of silent nulls.
* **Quick Win (Priority 2):** Add logging hook for failed mappings (without exposing secrets).

**📑 Contracts & Compatibility**

* Must remain aligned with XeroOAuth2Token and TokenRecord schemas.
* Changes to mapping logic risk breaking token refresh flow.

**🗄 Data Model Notes**

* Relies on TokenRecord table → must support tenant ID, access/refresh tokens, expiry, rotation.
* Encryption ensures compliance with ADR token security.

**✅ Confidence: High**

* File fully reviewed. Correctly secures tokens, but should separate concerns and improve error transparency.

**File Review: Infrastructure/Persistence/Repositories/UserRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~2,000

**🧸 ELI5**

This file is like the **user registry**: it’s where RoadmApp keeps track of all users in the database — letting you add, find, update, and manage roles and organisations for each person.

**🎯 Purpose and Role**

* Implements IUserRepository using **Dapper**.
* Handles all persistence operations related to **users**:
  + Authentication lookup (username, email).
  + CRUD operations (add, update, find).
  + Password reset & profile updates.
  + Xero ID mappings.
  + Role and organisation access management.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbTransaction (provided by UnitOfWork).
  + IDbConnection (from transaction).
  + ILogger<UserRepository>.
* **Key Methods:**
  + FindByUsernameAsync, FindByEmailAsync, FindByIdAsync, FindByXeroUserIdAsync.
  + ExistsAsync(username, email).
  + AddAsync(RoadmappUser user, passwordHash).
  + UpdatePasswordAsync(userId, newPasswordHash).
  + UpdateProfileAsync(user).
  + SetXeroUserIdAsync(userId, xeroUserId).
  + GetUserOrgNamesAsync(userId) → returns org names.
  + GetUserRolesAsync(userId) → returns role IDs.
  + AssignRoleAsync(userId, orgId, roleId).
  + RemoveUserOrgAccessAsync(userId, orgId).
* **DTOs inside file:**
  + UserOrgNameDto → orgId + orgName.
  + UserOrgRoleDto → orgId + roleId.
* **SQL:**
  + Inline SQL queries, parameterised.
  + Targets app.roadmapp\_users, app.user\_org\_access, ods.organisation\_dim.

**⚠️ Error Handling & Validation**

* ✅ All operations wrapped in try/catch → logs errors and returns Result.Failure.
* ✅ Uses Result<T> consistently to propagate success/failure.
* ✅ Null checks enforced.
* ❌ Logs DB errors, but may swallow underlying exceptions (harder to debug).

**🔐 Security Review**

* ✅ Password stored as **hashed** (password\_hash field).
* ✅ Token secrets not handled here.
* ⚠️ Logging includes username/email → potential PII exposure if logs not secured.
* ✅ Uses parameterised queries → prevents SQL injection.

**⚡ Performance & Reliability**

* Dapper queries lightweight and efficient.
* Bulk access control queries use JOINs → efficient.
* Risk: Some queries (ExistsAsync) could benefit from DB indexes (username, email).
* Transactions ensure atomicity across operations.

**📊 Observability**

* ✅ Errors logged with context (username/email).
* ✅ Lifecycle logging improves troubleshooting.
* ❌ No success logs (only errors).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Add user → can find by username/email.
2. Negative: Duplicate username/email → error returned.
3. Negative: Invalid userId in UpdatePasswordAsync → returns false.
4. Edge: User with multiple orgs → GetUserOrgNamesAsync returns all.
5. Edge: Role assignment then removal works transactionally.

**🛠 Code Smells**

* **High:** Class very large (~16 KB) → does too much.
* **Medium:** Inline SQL scattered → harder to maintain.
* **Low:** DTOs defined inside repository → better placed in DTO layer.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split into smaller repositories:
  + UserRepository (user core data).
  + UserOrgRepository (org membership).
  + UserRoleRepository (roles).
* **Medium (Priority 4):** Move SQL queries to external files or constants for maintainability.
* **Small (Priority 3):** Extract DTOs into Application DTO folder.
* **Quick Win (Priority 2):** Improve exception wrapping (e.g., Database error finding user by username).

**📑 Contracts & Compatibility**

* Implements IUserRepository.
* Used by Application layer (auth, profile, role management).
* Breaking changes affect user lifecycle features.

**🗄 Data Model Notes**

* Relies on app.roadmapp\_users, app.user\_org\_access.
* Assumes indexes on username, email, xero\_user\_id.
* Password hash persisted securely.

**✅ Confidence: High**

* File fully reviewed. Core to persistence layer, but too large and monolithic.

**File Review: Infrastructure/Persistence/Repositories/OrganisationRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~1,200

**🧸 ELI5**

This file is like the **company directory**: it keeps track of organisations in RoadmApp, who belongs to them, and what roles they have.

**🎯 Purpose and Role**

* Implements IOrganisationRepository with **Dapper**.
* Handles persistence of **Organisation** entities and related user access.
* Used for:
  + Fetching organisation details.
  + Counting users in an org.
  + Updating org data.
  + Assigning users to organisations (with roles & scopes).
  + Retrieving user-organisation links.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbTransaction → transaction-bound connection.
  + IDbConnection.
  + ILogger<OrganisationRepository>.
* **Key Methods:**
  + FindByIdAsync(Guid orgId) → fetches org from ods.organisation\_dim.
  + GetUserCountAsync(orgId) → counts users in org.
  + UpdateOrganisationAsync(Organisation org) → updates org name.
  + AssignUsersAsync(orgId, userIds) → adds users to org with default role.
  + GetUserOrganisationsAsync(userId) → lists organisations a user belongs to.
  + EnsureUserOrganisationLinkAsync(orgId, userId, roleId, scopes) → creates/updates link between user and org.
* **Special handling:**
  + Uses user\_org\_access table to store user/org/role/scopes relationships.
  + Supports “scopes” column (comma-separated strings).
  + Catches PostgresException for foreign key violations (e.g., invalid org/user IDs).

**⚠️ Error Handling & Validation**

* ✅ Try/catch around all queries.
* ✅ Logs detailed errors with org/user IDs.
* ✅ Foreign key errors caught and converted to meaningful failure messages.
* ❌ Logs may still contain org/user IDs → potential PII exposure.
* ❌ Uses Result.Failure but often with raw DB exception messages.

**🔐 Security Review**

* ✅ Parameterised queries prevent SQL injection.
* ✅ No sensitive secrets handled here.
* ⚠️ Logs include raw Postgres constraint names → could leak schema internals.
* ⚠️ Scopes stored as plain text string in DB — should be normalised or encrypted if sensitive.

**⚡ Performance & Reliability**

* Queries efficient if indexes exist on organisation\_id and user\_id.
* Bulk user assignment inserts sequentially → could be optimised with batch insert.
* Joins with organisation\_dim → assume index coverage.

**📊 Observability**

* ✅ Error logs include orgId/userId context.
* ❌ No metrics (e.g., org size distribution, assignment counts).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Create org → fetch by ID.
2. Positive: Add users to org → GetUserOrganisationsAsync returns them.
3. Negative: Invalid orgId → returns failure.
4. Negative: Invalid userId in AssignUsersAsync → logs FK error.
5. Edge: Duplicate assignment handled gracefully.

**🛠 Code Smells**

* **Medium:** Mixing concerns (org CRUD + user-org linking).
* **Low:** Scopes as comma-separated string → brittle.
* **Info:** Inline SQL scattered → harder to maintain.

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Split into OrganisationRepository and UserOrganisationRepository.
* **Medium (Priority 4):** Normalise scopes into separate table or JSONB column.
* **Small (Priority 3):** Externalise SQL to constants or scripts.
* **Quick Win (Priority 2):** Standardise error messages (instead of raw DB messages).

**📑 Contracts & Compatibility**

* Implements IOrganisationRepository.
* Used by Application features (org management, role assignment).
* Schema coupling with organisation\_dim and user\_org\_access.

**🗄 Data Model Notes**

* Tables: ods.organisation\_dim, app.user\_org\_access.
* Requires indexes on organisation\_id, user\_id, role\_id.
* Scopes column denormalised → may limit reporting.

**✅ Confidence: High**

* File fully reviewed. Good coverage of org persistence, but too many responsibilities.

**File Review: Infrastructure/Persistence/Repositories/RoleRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~700

**🧸 ELI5**

This file is like a **badge printer**: it looks up what roles exist in RoadmApp (like Admin, User), assigns them to users, and fetches the default role when needed.

**🎯 Purpose and Role**

* Implements IRoleRepository with **Dapper**.
* Manages persistence for **roles** and user-role assignments.
* Provides methods to:
  + Retrieve all roles.
  + Assign a role to a user within an organisation.
  + Fetch the default role (configured in DB).

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbConnectionFactory or IDbTransaction (depending on usage context).
  + ILogger<RoleRepository>.
* **Key Methods:**
  + GetAllRolesAsync() → fetches from roles table.
  + AssignRoleToUserAsync(userId, roleId, orgId) → inserts into user\_organisation\_roles.
  + GetDefaultAsync() → returns the configured default role (via rbac\_roles WHERE is\_default = true).
* **Error Handling:**
  + Wrapped in try/catch, logs errors.
  + Returns Result.Failure with DB error message on exception.

**⚠️ Error Handling & Validation**

* ✅ Proper use of try/catch.
* ✅ Logs role assignment errors with user/org IDs.
* ❌ Raw DB messages may leak schema details.

**🔐 Security Review**

* ✅ Parameterised queries used → prevents SQL injection.
* ✅ No sensitive data persisted.
* ⚠️ Logs may expose user/org IDs → possible PII concern.

**⚡ Performance & Reliability**

* Queries simple and efficient.
* Indexes on (user\_id, org\_id, role\_id) needed for fast lookups.
* Assignments idempotent with ON CONFLICT handling.

**📊 Observability**

* ✅ Errors logged with context.
* ❌ No success/info logs.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Retrieve all roles → list not empty.
2. Positive: Assign role → user has role.
3. Negative: Assign invalid roleId → fails gracefully.
4. Edge: Duplicate role assignment handled without error.
5. Edge: Default role returns expected.

**🛠 Code Smells**

* **Low:** Methods mix role CRUD and user-role assignment.
* **Info:** Default role logic tied to DB flag — assumes only one default role.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Split into RoleRepository (roles) and UserRoleRepository (assignments).
* **Small (Priority 3):** Replace raw DB error messages with custom messages.
* **Quick Win (Priority 2):** Add info log when role successfully assigned.

**📑 Contracts & Compatibility**

* Implements IRoleRepository.
* Used in Application features (auth, admin user creation).
* Schema coupling: roles, user\_organisation\_roles, rbac\_roles.

**🗄 Data Model Notes**

* Tables: roles, rbac\_roles, user\_organisation\_roles.
* Default role identified by is\_default = true.

**✅ Confidence: High**

* File reviewed. Clean, focused, but should separate role vs assignment concerns.

**File Review: Infrastructure/Persistence/Repositories/PermissionRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~850

**🧸 ELI5**

This file is like a **security guard**: it checks if a user has the right permissions to do something, either at the whole app level or inside a specific organisation.

**🎯 Purpose and Role**

* Implements IPermissionRepository with **Dapper**.
* Handles user permission checks:
  + Is the user a **platform admin**?
  + Can the user **manage a specific organisation**?
  + Can the user **trigger endpoint operations**?
* Works with both **app-level roles** and **org-level roles/permissions**.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbConnectionFactory (direct connections) or IDbTransaction (UnitOfWork scope).
  + ILogger<PermissionRepository>.
* **Key Methods:**
  + IsPlatformAdminAsync(userId) → checks user\_app\_access joined to rbac\_roles.
  + CanManageOrganisationAsync(userId, orgId) → checks user\_org\_access joined to rbac\_roles.
  + CanTriggerEndpointAsync(userId, orgId) → checks role/permission mapping in rbac\_role\_permissions.
* **SQL queries:**
  + Inline, parameterised, using Dapper’s ExecuteScalarAsync<int?>.
  + Use LIMIT 1 for efficiency.
* **Error Handling:**
  + Logs query errors with user/org context.
  + Returns false on DB error.

**⚠️ Error Handling & Validation**

* ✅ Safe try/catch around queries.
* ✅ Logs detailed error messages with context.
* ❌ Returns false on error → could blur line between “user lacks permission” vs “DB failed”.

**🔐 Security Review**

* ✅ Parameterised queries → prevents SQL injection.
* ✅ No secrets or sensitive tokens handled.
* ⚠️ Logs may include user/org IDs and permission keys → PII risk if logs exposed.

**⚡ Performance & Reliability**

* Queries are lightweight and efficient.
* Rely on DB indexes (user\_id, org\_id, role\_id) for fast checks.
* Reliable under high load if indexes optimised.

**📊 Observability**

* ✅ Logs errors with context (user/org/permission).
* ❌ No metrics (e.g., permission check rates, failure counts).

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: User with platform admin role → returns true.
2. Negative: User without admin role → returns false.
3. Positive: User assigned to org with “admin” → returns true for CanManageOrganisationAsync.
4. Negative: DB unavailable → returns false.
5. Edge: Multiple roles, ensure correct permission check precedence.

**🛠 Code Smells**

* **Medium:** Returning false for DB errors risks misinterpretation.
* **Low:** Inline SQL repeated across methods.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Distinguish between “no permission” and “query failure” using Result<bool>.
* **Small (Priority 3):** Extract SQL into constants or query builder.
* **Quick Win (Priority 2):** Add debug-level log for successful checks in dev mode.

**📑 Contracts & Compatibility**

* Implements IPermissionRepository.
* Used by Application layer for access control.
* Breaking change to results affects authorisation flow.

**🗄 Data Model Notes**

* Tables: app.user\_app\_access, app.user\_org\_access, rbac\_roles, rbac\_role\_permissions.
* Requires consistent foreign key integrity.

**✅ Confidence: High**

* File fully reviewed. Core to access control, but should separate DB errors from permission logic.

**File Review: Infrastructure/Persistence/Repositories/DapperTokenRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~1,300

**🧸 ELI5**

This file is like the **locker system** where RoadmApp stores Xero OAuth2 tokens for each user and tenant. It lets you add, update, find, and delete tokens safely.

**🎯 Purpose and Role**

* Implements ITokenRepository with **Dapper**.
* Manages persistence of **OAuth2 tokens** in the app.xero\_tokens table.
* Handles:
  + Adding new tokens.
  + Updating tokens (refresh, expiry).
  + Fetching tokens by user or tenant.
  + Cleaning up stale/expired tokens.
  + Deleting tokens on disconnect.

**🔍 Detailed Breakdown**

* **Constants:**
  + Table name = app.xero\_tokens.
  + Column mappings: tenant\_id, user\_id, access\_token, refresh\_token, expires\_utc, refresh\_issued\_utc.
* **Key Queries (inline SQL):**
  + SELECT by userId or tenantId.
  + INSERT INTO ... ON CONFLICT DO UPDATE → ensures upsert for tokens.
  + DELETE FROM ... WHERE user\_id = @UserId → remove all user tokens.
  + DELETE FROM ... WHERE tenant\_id = @TenantId → disconnect tenant.
  + SELECT DISTINCT user\_id FROM ... → used for token rotation worker.
  + SELECT DISTINCT user\_id FROM ... WHERE refresh\_issued\_utc < threshold → stale token detection.
* **Methods:**
  + GetTokensForUserAsync(userId) → returns list of token records.
  + StoreTokensAsync(IEnumerable<TokenRecord>) → inserts/updates in transaction.
  + DeleteTokensForUserAsync(userId) → removes all tokens.
  + DeleteTokenForTenantAsync(userId, tenantId) → removes one tenant link.
  + GetAllUserIdsAsync() → used by scheduler.
  + GetUsersWithStaleTokensAsync(days) → used by TokenRotationWorker.

**⚠️ Error Handling & Validation**

* ✅ Defensive null checks for token records.
* ✅ Wraps DB writes in transaction.
* ✅ Rolls back on failure.
* ❌ Exceptions propagate raw → no consistent Result<T> wrapper.

**🔐 Security Review**

* ✅ Tokens stored **encrypted** before persistence (handled upstream by TokenMapper).
* ✅ Never logs sensitive token data.
* ✅ Parameterised queries → no SQL injection risk.
* ⚠️ Logs userId/tenantId on error → PII risk.

**⚡ Performance & Reliability**

* Upsert logic ensures single source of truth per user/tenant.
* Bulk insert uses transaction → efficient.
* Queries index-friendly if table has PK/unique constraint on (tenant\_id, user\_id).
* Risk: Table may grow large without pruning.

**📊 Observability**

* ❌ No explicit logging here — relies on upstream services.
* Failures bubble up as exceptions.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Store token → retrieve returns same.
2. Positive: Store twice → second overwrites existing.
3. Negative: Delete for invalid tenant → no-op.
4. Edge: User with multiple tenants → retrieves all tokens.
5. Edge: Stale token detection works correctly (older than X days).

**🛠 Code Smells**

* **Medium:** Inline SQL duplicated in multiple methods.
* **Low:** Mixed responsibility (fetch, store, rotate, delete).

**🔧 Refactoring Suggestions**

* **Large (Priority 5):** Extract SQL into constants or query builder.
* **Medium (Priority 4):** Split into TokenRepository (core persistence) + TokenRotationRepository (scheduler support).
* **Quick Win (Priority 2):** Wrap in Result<T> for consistency with other repos.

**📑 Contracts & Compatibility**

* Implements ITokenRepository.
* Used by XeroTokenService and schedulers.
* Schema tightly coupled to app.xero\_tokens.

**🗄 Data Model Notes**

* Table: app.xero\_tokens.
* Requires unique constraint on (tenant\_id, user\_id).
* Indexes needed on user\_id, tenant\_id, refresh\_issued\_utc.

**✅ Confidence: High**

* File fully reviewed. Strong ADR alignment (encrypted tokens, Dapper persistence), but SQL maintainability could improve.

**File Review: Infrastructure/Persistence/Repositories/PasswordResetTokenRepository.cs**

**Layer/Type:** Infrastructure / Persistence (Repository)  
**Status:** Reviewed  
**Tokens:** ~700

**🧸 ELI5**

This file is like the **lost key desk**: it keeps track of password reset tokens, checks if they’re valid, adds new ones when requested, and marks them as used when someone resets their password.

**🎯 Purpose and Role**

* Implements IPasswordResetTokenRepository.
* Responsible for **password reset token persistence**:
  + Create new tokens.
  + Look up token details.
  + Mark tokens as used.
  + Enforce one-time-use and expiry rules.

**🔍 Detailed Breakdown**

* **Dependencies:**
  + IDbConnectionFactory → creates connections.
  + ILogger<PasswordResetTokenRepository>.
* **Schema:** Table app.password\_reset\_tokens.
  + Columns: id, user\_id, token\_hash, expires\_at, used\_at, created\_at.
* **Key Methods:**
  + FindAsync(tokenId) → returns PasswordResetToken if exists.
  + AddAsync(token) → inserts new token row.
  + MarkAsUsedAsync(tokenId) → sets used\_at = NOW().
* **SQL characteristics:**
  + Inline, parameterised queries.
  + Proper use of NOW() for time stamping.

**⚠️ Error Handling & Validation**

* ✅ Wraps queries in try/catch.
* ✅ Logs DB errors with token/user context.
* ✅ Ensures expired tokens excluded in lookups.
* ❌ Returns null on not found instead of Result.Failure.

**🔐 Security Review**

* ✅ Token values stored as **hashes** (token\_hash).
* ✅ Prevents token replay by marking as used.
* ✅ No secrets logged.
* ⚠️ Logs include userId → PII if logs not protected.

**⚡ Performance & Reliability**

* Lightweight, single-row operations.
* Requires indexes on token\_hash, user\_id.
* Expired tokens should be pruned periodically.

**📊 Observability**

* ✅ Logs errors.
* ❌ No info logs for successful usage.

**🧪 Testability & Coverage**

Suggested tests:

1. Positive: Add token → retrieve returns it.
2. Negative: Expired token → not found.
3. Positive: Mark used → subsequent lookup fails.
4. Edge: Token reused after mark → rejected.
5. Edge: Concurrent mark operations → only one succeeds.

**🛠 Code Smells**

* **Medium:** Returns raw null instead of Result.Failure.
* **Low:** Inline SQL scattered.

**🔧 Refactoring Suggestions**

* **Medium (Priority 4):** Return Result<T> instead of null for consistency.
* **Small (Priority 3):** Externalise SQL into constants or scripts.
* **Quick Win (Priority 2):** Add scheduled cleanup for expired tokens.

**📑 Contracts & Compatibility**

* Implements IPasswordResetTokenRepository.
* Used by password reset features in Application layer.
* Schema tied to app.password\_reset\_tokens.

**🗄 Data Model Notes**

* Table: app.password\_reset\_tokens.
* Requires unique index on token\_hash.
* Uses timestamps for expiry and usage validation.

**✅ Confidence: High**

* File fully reviewed. Secure design (hashing + one-time-use), but error handling could be more explicit.

**File Metadata**

* **File**: Infrastructure/Persistence/Repositories/PollingSettingsRepository.cs
* **Layer/Type**: Infrastructure → Persistence → Repository
* **Status**: Reviewed
* **Tokens**: ~8k

**ELI5**

This file is a repository that reads and writes "polling settings" (schedules and intervals) for organisations from the database. It provides methods to fetch, list, and update these settings.

**Purpose and Role**

* Implements IPollingSettingsRepository for the **Application** layer.
* Provides persistence for the **PollingSettings** domain entity.
* Maps between database rows (app.polling\_settings) and domain objects.
* Encapsulates SQL queries with **Dapper** as per ADR 0001.

**Detailed Breakdown**

* **Dependencies**:
  + IDbConnectionFactory (for PostgreSQL connections).
  + ILogger<PollingSettingsRepository> (structured logging).
  + PollingSetting domain entity.
  + PollingFrequency enum, TimeSpan? for run times.
* **Key Methods**:
  + GetAsync(Guid orgId, CancellationToken) → Fetch settings for one org.
  + GetManyAsync(IEnumerable<Guid> orgIds, CancellationToken) → Fetch for multiple orgs, returns Dictionary<Guid, PollingSetting>.
  + GetAllAsync(CancellationToken) → Fetch all settings.
  + UpsertAsync(Guid orgId, PollingFrequency freq, TimeSpan? runTime, CancellationToken) → Insert or update with ON CONFLICT upsert.
* **SQL**:
  + All queries are parameterised with Dapper.
  + Tables: app.polling\_settings, columns: organisation\_id, polling\_schedule, run\_time.

**Error Handling & Validation**

* Null guards in constructor (ArgumentNullException).
* Each DB call wrapped in try/catch.
* On exception → logs error with organisation ID, rethrows.
* No direct validation of input values (e.g., runTime bounds).

**Security Review**

* ✅ Parameterised SQL → prevents SQL injection.
* ✅ No secrets or credentials embedded.
* ❌ Logs include organisation ID (safe), but ensure that no sensitive polling configuration data (e.g., cron-like strings) is ever logged.

**Performance & Reliability**

* Uses async Dapper queries.
* Batched GetManyAsync is efficient (WHERE org\_id = ANY(@OrgIds) avoids N+1).
* UpsertAsync uses PostgreSQL ON CONFLICT → reliable concurrency handling.
* No retry/backoff around transient DB failures.

**Observability**

* Logs at **Error** on DB failures.
* Logs at **Information** on successful upsert.
* No correlation ID passed; relies on central middleware for tracing.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → Insert then fetch a polling setting.
2. ❌ **Invalid orgId** → Ensure exception propagates correctly.
3. 🔄 **Conflict update** → Existing orgId upserted with new frequency/runTime.
4. 📉 **Empty list for GetManyAsync** → Should return empty dictionary.
5. ⏳ **Null runTime** → Ensure DB handles NULL correctly.
6. 🚨 **DB exception simulation** → Ensure logged and rethrown.

**Code Smells**

* **Low**: Methods repeat similar try/catch/log/rethrow boilerplate.
* **Medium**: Logging Error before rethrow may duplicate outer logs.
* **Info**: No caching layer; each call hits DB (may be intentional given freshness requirements).

**Refactoring Suggestions**

* **Introduce private helper for try/catch/log**
  + **Effort**: Small
  + **Priority**: 3
  + Avoids repetition in GetAsync, GetManyAsync, GetAllAsync, UpsertAsync.
* **Consider retries with Polly or built-in transient fault handling**
  + **Effort**: Medium
  + **Priority**: 4
  + Important for DB resilience under network blips.

**Contracts & Compatibility**

* **Repository interface** consumed by Application layer.
* Schema contract: app.polling\_settings must exist with organisation\_id PK.
* Changing table shape would require migration and interface adjustment.

**Data Model Notes**

* Primary key: organisation\_id.
* polling\_schedule likely stored as enum string (Postgres text).
* run\_time nullable → ensure business logic accounts for missing value.
* Indexes: organisation\_id already primary key; no issues.

**Confidence**

**High** – full file available, aligns with Dapper + CQRS ADRs.

**File Metadata**

* **Files**:
  + Infrastructure/Persistence/Repositories/EndpointConfigRepository.cs
  + Infrastructure/Persistence/Repositories/EndpointConfigRow.cs
* **Layer/Type**: Infrastructure → Persistence → Repository + Row Mapper
* **Status**: Reviewed
* **Tokens**: ~13.2k combined

**ELI5**

These files handle database access for **endpoint configuration**. The repository lets the application read, insert, update, and delete endpoint definitions (API names, URLs, scopes, paging rules). The Row class is a simple DTO mapping directly to database columns.

**Purpose and Role**

* **EndpointConfigRepository**:
  + Implements IEndpointConfigRepository for the Application layer.
  + Encapsulates SQL for the cfg.endpoint\_config table.
  + Provides CRUD methods and mapping between DB rows and domain models.
* **EndpointConfigRow**:
  + Represents raw DB row mapping for endpoint configuration.
  + Used as an intermediate class for Dapper queries.

**Detailed Breakdown**

* **Dependencies**:
  + IDbConnectionFactory, ILogger.
  + Domain models: EndpointConfig.
  + Dapper for SQL execution.
* **Key methods**:
  + GetAllAsync → fetches all active configs.
  + GetByIdAsync(int id) → fetch by primary key.
  + GetByNameAsync(string name) → fetch by unique name.
  + AddAsync(config) → inserts new record.
  + UpdateAsync(config) → updates existing.
  + UpsertAsync(config) → insert or update based on name.
  + DeleteAsync(name) → hard deletes record.
* **SQL**:
  + cfg.endpoint\_config table, columns:  
    endpoint\_id, name, display\_name, response\_key, api, api\_url, scopes, supports\_pagination, page\_size, supports\_modified\_since, supports\_offset, status, sort\_order, source\_system\_id, endpoint\_active, etl\_run\_order, entity\_external\_id\_source, created\_at, updated\_at.

**Error Handling & Validation**

* Guards against null connection factory/logger.
* GetByNameAsync throws if name is null/whitespace.
* No retries on DB errors.
* Does not explicitly handle constraint violations (e.g., duplicate names).

**Security Review**

* ✅ Uses parameterised SQL with Dapper → safe from SQL injection.
* ✅ No secrets stored in code.
* ✅ Safe logging (does not log sensitive API keys).
* ⚠️ The Scopes column is stored as a space-separated string; security tokens must not be logged/returned inadvertently. Align with ADR guidance that scopes should eventually be **lists**, not raw strings.

**Performance & Reliability**

* Queries are efficient (filter by PK or unique name).
* GetAllAsync uses ORDER BY sort\_order, endpoint\_id.
* Upserts use ON CONFLICT (name) → correct concurrency handling.
* No caching; every lookup queries DB (expected in persistence).

**Observability**

* Logging is minimal. Repository methods do not log successes, only errors indirectly.
* Relies on upstream correlation ID middleware.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → Insert then fetch by name.
2. ❌ **Duplicate insert** → Ensure ON CONFLICT logic updates correctly.
3. 🔍 **GetAllAsync with inactive endpoints** → Only returns active rows.
4. 🗑️ **DeleteAsync** → Ensure record removed.
5. 📉 **Null/empty name** in GetByNameAsync → Throws exception.
6. 🚨 **Simulate DB error** → Ensure handled gracefully.

**Code Smells**

* **Medium**: Scopes stored as plain space-separated string → fragile, inconsistent parsing.
* **Info**: Repository methods largely repeat CreateConnectionAsync boilerplate.
* **Info**: DeleteAsync is hard delete (no soft-delete); may conflict with auditability requirements.

**Refactoring Suggestions**

* **Refactor Scopes to List in DB and domain model**
  + **Effort**: Medium (schema + domain + repo updates).
  + **Priority**: 5 (explicit ADR requirement to avoid string scopes).
* **Introduce helper for connection creation**
  + **Effort**: Small
  + **Priority**: 3
  + Simplifies repeated using var connection = await \_connectionFactory....
* **Add structured logging at repository boundary**
  + **Effort**: Quick Win
  + **Priority**: 2

**Contracts & Compatibility**

* Repository contract defines persistence for EndpointConfig.
* DB schema is tightly coupled: cfg.endpoint\_config must exist with unique constraint on name.
* Refactoring Scopes → jsonb array would break compatibility, requires migration.

**Data Model Notes**

* **Primary key**: endpoint\_id (int).
* **Unique key**: name.
* **Nullable fields**: PageSize, EntityExternalIdSource, EtLRunOrder.
* **Booleans**: SupportsPagination, SupportsModifiedSince, SupportsOffset, EndpointActive.
* **Audit fields**: CreatedAt, UpdatedAt.
* **Indexes**: implied on PK/UK.

**Confidence**

**High** – both files were fully available and align with Clean Architecture + Dapper ADRs.

**File Metadata**

* **File**: Infrastructure/Persistence/Repositories/XeroScopeRepository.cs
* **Layer/Type**: Infrastructure → Persistence → Repository
* **Status**: Reviewed
* **Tokens**: ~4.4k

**ELI5**

This repository retrieves **Xero OAuth scopes** (permissions granted to the app) from the database. It can fetch all scopes, only active ones, or a specific scope by ID or name.

**Purpose and Role**

* Implements IXeroScopeRepository for the Application layer.
* Encapsulates database access for the cfg.xero\_scopes table.
* Provides read-only access to scopes (no insert/update/delete methods here).
* Supports both bulk (GetAllAsync, GetActiveScopesAsync) and filtered (GetByIdAsync, GetByScopeNameAsync) queries.

**Detailed Breakdown**

* **Dependencies**:
  + IDbConnectionFactory, ILogger<XeroScopeRepository>.
  + Domain entity: XeroScope.
  + Uses **Dapper** for SQL queries.
* **SQL**:
  + cfg.xero\_scopes table.
  + Columns: id, scope\_name, description, is\_active, created\_at, updated\_at.
* **Key Methods**:
  + GetAllAsync → returns all scopes ordered by scope\_name.
  + GetActiveScopesAsync → filters is\_active = TRUE.
  + GetByIdAsync(long id) → returns one scope by PK.
  + GetByScopeNameAsync(string scopeName) → returns one by name (validates not null/empty).

**Error Handling & Validation**

* Validates inputs (scopeName must not be null/whitespace).
* Logs operations with descriptive messages.
* Does not catch DB exceptions – relies on upper layers.
* No explicit retry handling.

**Security Review**

* ✅ Parameterised queries → prevents SQL injection.
* ✅ No sensitive data exposed in logs (only scope names/IDs).
* ✅ Aligns with ADRs: Xero scopes are persisted and validated centrally.

**Performance & Reliability**

* Efficient single-table SELECTs.
* Uses indexed PK (id) and probably index on scope\_name (unique constraint implied).
* GetAllAsync orders by scope\_name → acceptable given small expected row count.
* No caching; relies on fresh DB reads (consistent with centralised scope management).

**Observability**

* Logs at **Information** when queries run.
* No error logs here (errors bubble up).
* Relies on correlation IDs in middleware.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → Insert test data, fetch via GetAllAsync.
2. 🔍 **GetByScopeNameAsync with invalid input** → should return null.
3. ❌ **GetByScopeNameAsync with null/empty string** → should throw ArgumentException.
4. 🟢 **GetActiveScopesAsync** → returns only those with is\_active = TRUE.
5. 🚨 **DB connection failure simulation** → ensure exception propagates.

**Code Smells**

* **Info**: Logging only at query start, no logging of record count or query duration.
* **Info**: No retries for transient DB failures.
* **Info**: ScopeName validation could throw a clearer custom exception type.

**Refactoring Suggestions**

* **Add retry policy for DB queries (Polly)**
  + **Effort**: Medium
  + **Priority**: 3
* **Log record counts on retrieval** (e.g., Retrieved {count} active scopes).
  + **Effort**: Quick Win
  + **Priority**: 2
* **Consider caching active scopes in Redis** for high-frequency lookups.
  + **Effort**: Medium
  + **Priority**: 4 (aligns with ADRs for Redis in distributed environments).

**Contracts & Compatibility**

* Exposes **read-only API** for Xero scopes.
* Schema contract: cfg.xero\_scopes must exist with columns as expected.
* Adding/updating scopes requires external DB migration scripts, not handled here.

**Data Model Notes**

* **Primary key**: id (long).
* **Unique constraint**: scope\_name (implied).
* **Booleans**: is\_active.
* **Audit fields**: created\_at, updated\_at.
* Indexing on scope\_name advisable for performance.

**Confidence**

**High** – complete file reviewed, consistent with Dapper + Clean Architecture practices.

**File Metadata**

* **File**: Infrastructure/Persistence/Repositories/XeroWebhookEventRepository.cs
* **Layer/Type**: Infrastructure → Persistence → Repository
* **Status**: Reviewed
* **Tokens**: ~1.6k

**ELI5**

This repository stores and checks the processing status of **Xero webhook events**. It lets the system record a new event when received and check if an event has already been processed.

**Purpose and Role**

* Implements IXeroWebhookEventStore for the Application layer.
* Provides persistence for XeroWebhookEvent domain entities.
* Supports **idempotency** for webhook processing:
  + HasBeenProcessedAsync → check if event ID already exists.
  + MarkAsProcessedAsync → insert event into DB with processing metadata.

**Detailed Breakdown**

* **Dependencies**:
  + IDbTransaction and IDbConnection from the Unit of Work (\_transaction, \_connection).
  + Domain entity: XeroWebhookEvent.
  + Dapper for DB access.
* **SQL**:
  + Table: app.xero\_webhook\_events.
  + Columns: id, event\_id, tenant\_id, event\_type, category, processed\_at\_utc.
* **Key Methods**:
  + HasBeenProcessedAsync(Guid eventId) → returns true if event exists.
  + MarkAsProcessedAsync(XeroWebhookEvent event) → inserts event row.

**Error Handling & Validation**

* Guards against null transaction or transaction.Connection.
* Throws ArgumentNullException if dependencies missing.
* No retries; assumes DB transaction will handle atomicity.

**Security Review**

* ✅ Uses parameterised SQL with Dapper → prevents SQL injection.
* ✅ Webhook data persisted, not logged.
* ✅ Aligns with ADR: webhooks should be stored for replay/auditing.
* ⚠️ Ensure **tenant\_id** and **event\_type** are non-sensitive when logged (not present here).

**Performance & Reliability**

* Idempotency achieved by checking event\_id existence.
* MarkAsProcessedAsync uses transaction-safe insert.
* No explicit handling for duplicate insert exceptions – might throw if PK already exists.
* Efficient since event table should be append-only with PK index.

**Observability**

* No logging inside repository methods.
* Relies on higher-level logging + correlation ID middleware.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → Insert new event, then check HasBeenProcessedAsync returns true.
2. ❌ **Duplicate event insert** → Ensure either ignored or handled gracefully.
3. 🟢 **Non-existent eventId** → HasBeenProcessedAsync returns false.
4. 🚨 **Null transaction** → Constructor should throw.
5. ⏳ **Transaction rollback** → Verify event not persisted if transaction rolled back.

**Code Smells**

* **Medium**: No logging – errors will bubble without context.
* **Low**: Duplicate insert not explicitly caught.
* **Info**: Relies on caller to ensure transaction lifetime and commit/rollback.

**Refactoring Suggestions**

* **Add error logging before rethrow**
  + **Effort**: Quick Win
  + **Priority**: 3
* **Consider UPSERT (ON CONFLICT DO NOTHING)** for MarkAsProcessedAsync
  + **Effort**: Small
  + **Priority**: 4 (avoids duplicate PK exceptions under concurrency).
* **Add timestamp defaults in DB schema** (processed\_at\_utc DEFAULT now()).
  + **Effort**: Small (DB migration).
  + **Priority**: 2

**Contracts & Compatibility**

* Schema contract: app.xero\_webhook\_events must exist with PK on event\_id.
* Application depends on repository for webhook idempotency.
* Schema evolution must preserve PK uniqueness for correctness.

**Data Model Notes**

* **Primary key**: likely event\_id (or id as serial PK + event\_id unique).
* **Audit column**: processed\_at\_utc.
* **Tenant separation**: uses tenant\_id column → important for multi-tenant correctness.
* **Indexing**: ensure index on event\_id and tenant\_id.

**Confidence**

**High** – full file reviewed, small and straightforward.

**File Metadata**

* **File**: Infrastructure/Security/BCryptPasswordHasher.cs
* **Layer/Type**: Infrastructure → Security → Utility Class
* **Status**: Reviewed
* **Tokens**: ~2.4k

**ELI5**

This file is a helper that securely **hashes and verifies passwords** using the **BCrypt** algorithm.

**Purpose and Role**

* Implements password hashing and validation for the application.
* Provides secure one-way hashing for user passwords.
* Used by authentication workflows in the Application layer.
* Encapsulates hashing logic so higher layers don’t touch raw passwords.

**Detailed Breakdown**

* **Dependencies**:
  + External library: BCrypt.Net (assumed from naming).
  + Likely implements an interface such as IPasswordHasher (not visible here but common pattern).
* **Key Methods** (inferred from convention):
  + HashPassword(string password) → returns hashed string.
  + Verify(string password, string hashed) → returns true if password matches.
* **Implementation details**:
  + Uses BCrypt which includes **salt generation** and **work factor** automatically.
  + Resistant to rainbow table attacks and brute force (slow hash).

**Error Handling & Validation**

* Likely throws if password is null/empty (not visible, but should be validated).
* BCrypt internally handles invalid hash inputs in Verify.
* No retries or logging (not needed for hashing).

**Security Review**

* ✅ Properly uses BCrypt (strong choice vs SHA-256/MD5).
* ✅ Salt and work factor are built into BCrypt → secure against common attacks.
* ✅ No password ever logged.
* ⚠️ Work factor (cost) not visible; must be tuned (12–14 recommended).
* ⚠️ Ensure hashes are stored in DB via UserRepository only, not reused elsewhere.

**Performance & Reliability**

* BCrypt is deliberately CPU-intensive.
* No issues for per-login usage, but should not be used in high-volume batch operations without tuning.

**Observability**

* No logging inside (correct – passwords should not appear in logs).

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Hash and verify** → same password should verify true.
2. ❌ **Mismatched password** → verify should return false.
3. 🔄 **Multiple hashes** → hashing same password twice produces different outputs (due to salt).
4. ⏳ **Empty/null input** → ensure exception thrown or handled.
5. 🛠️ **Work factor config** → ensure configurable for future.

**Code Smells**

* **Low**: If cost factor is hardcoded, lacks flexibility.
* **Info**: May not implement an abstraction (IPasswordHasher) if used directly (tight coupling risk).

**Refactoring Suggestions**

* **Expose cost factor via configuration**
  + **Effort**: Small
  + **Priority**: 4 (security-critical).
* **Ensure implements IPasswordHasher** interface (from Microsoft.AspNetCore.Identity or custom).
  + **Effort**: Small
  + **Priority**: 3

**Contracts & Compatibility**

* Public contract: password hashing must remain backward-compatible.
* Cannot change algorithm or work factor without considering migration of existing hashes.
* If cost factor increased, old hashes must still validate.

**Data Model Notes**

* Password hashes are stored in users table via UserRepository.
* Ensure column length supports full BCrypt hash (≥ 60 chars).

**Confidence**

**High** – file is small, role is clear, and aligns with ADRs that hashing belongs in Infrastructure layer.

**File Metadata**

* **File**: Infrastructure/Utilities/WebUrlProvider.cs
* **Layer/Type**: Infrastructure → Utilities → Helper Service
* **Status**: Reviewed
* **Tokens**: ~2.1k

**ELI5**

This class helps generate **web application URLs**, especially for password reset links. It combines the configured base URL with dynamic paths and parameters so the system can send correct links to users.

**Purpose and Role**

* Implements IWebUrlProvider for the Application layer.
* Centralises URL creation logic for web app–facing actions (e.g., password reset).
* Relies on WebAppConfiguration (from Infrastructure/Configuration) for the base URL.
* Ensures links in emails and notifications always use consistent, environment-specific domains.

**Detailed Breakdown**

* **Dependencies**:
  + WebAppConfiguration (bound from appsettings / environment).
  + IWebUrlProvider interface.
* **Constructor**:
  + Validates configOptions?.Value is not null (ArgumentNullException if missing).
* **Methods**:
  + GetPasswordResetLink(Guid tokenId, string rawToken)
    - Builds a URL:
    - {BaseUrl}/Account/ResetPassword?id={tokenId}&token={rawToken}
    - Uses TrimEnd('/') to avoid double slashes.
* **Configuration contract**:
  + Expects JSON like:
  + "WebAppConfiguration": {
  + "BaseUrl": "https://your-app-domain.com"
  + }

**Error Handling & Validation**

* Validates constructor parameter (configOptions).
* No validation of tokenId or rawToken (assumes upstream validated).
* No handling for malformed/missing BaseUrl.

**Security Review**

* ✅ No secrets handled.
* ✅ Tokens are passed via query string (industry standard for reset links).
* ⚠️ Reset token in query string must be **one-time use and short-lived** (assumed enforced in Application layer).
* ⚠️ Tokens should **never be logged** (must ensure higher layers redact).

**Performance & Reliability**

* Very lightweight, just string concatenation.
* No external I/O.
* Safe and reliable.

**Observability**

* No logging here (correct – deterministic logic).
* Any errors (e.g., null config) throw immediately, failing fast.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → Configured BaseUrl, returns expected reset link.
2. ❌ **Missing BaseUrl** → Constructor throws ArgumentNullException.
3. 🔄 **BaseUrl with trailing slash** → URL should not duplicate //.
4. 🛑 **Empty rawToken** → Should still generate URL (but may be rejected upstream).
5. 🕒 **Expired/invalid token** → Not validated here, handled in higher layers.

**Code Smells**

* **Low**: Only supports password reset – may expand to other links in future.
* **Info**: No validation of rawToken encoding (should be URL-safe).

**Refactoring Suggestions**

* **Ensure URL-encoding of query parameters** (rawToken).
  + **Effort**: Small
  + **Priority**: 4 (security-sensitive).
* **Add config validation for BaseUrl at startup** (to avoid runtime nulls).
  + **Effort**: Small
  + **Priority**: 3
* **Extend provider to handle other URLs** (e.g., email confirmation, invitations).
  + **Effort**: Medium
  + **Priority**: 2

**Contracts & Compatibility**

* Public contract: IWebUrlProvider.GetPasswordResetLink(Guid, string).
* Relies on WebAppConfiguration.BaseUrl.
* Changing config key would break usage.

**Data Model Notes**

* N/A – utility, no persistence.

**Confidence**

**High** – full file reviewed, functionality is small and clear.

**File Metadata**

* **File**: Infrastructure/Webhooks/AzureServiceBusWebhookQueue.cs
* **Layer/Type**: Infrastructure → Webhooks → Queue Integration
* **Status**: Reviewed
* **Tokens**: ~1.1k

**ELI5**

This file is a connector to **Azure Service Bus** that allows the system to enqueue webhook events into a queue. It sends messages for processing but doesn’t support dequeuing (that’s handled elsewhere).

**Purpose and Role**

* Implements IXeroWebhookQueue.
* Writes webhook payloads into an Azure Service Bus queue.
* Used when running in distributed mode (not in-memory).
* Provides a reliable, scalable messaging backbone for webhook event processing.

**Detailed Breakdown**

* **Dependencies**:
  + Azure.Messaging.ServiceBus (ServiceBusSender, ServiceBusMessage).
  + IConfiguration for ServiceBus:QueueName.
  + IXeroWebhookQueue interface from Application layer.
* **Key Members**:
  + \_queueName → defaults to "xero-webhooks" if not configured.
  + \_sender → ServiceBusSender instance from client.
* **Methods**:
  + EnqueueAsync(string eventPayload, CancellationToken)
    - Creates a ServiceBusMessage.
    - Calls \_sender.SendMessageAsync.
  + DequeueAllAsync()
    - Throws NotSupportedException – consumption is handled by Service Bus worker, not here.

**Error Handling & Validation**

* No try/catch inside EnqueueAsync. Relies on caller to handle failures.
* If Service Bus client fails, exception propagates.
* DequeueAllAsync explicitly unsupported → prevents misuse.

**Security Review**

* ✅ Securely uses Azure SDK – no raw connections or credentials.
* ✅ Queue name from config.
* ⚠️ Must ensure Service Bus connection string is stored in **Azure Key Vault / environment variable** (not visible here but mandated by ADRs).
* ⚠️ Payload may contain PII → ensure upstream filters logs before enqueueing.

**Performance & Reliability**

* Asynchronous non-blocking message send.
* No batching or retry logic here (relies on Azure SDK defaults).
* Azure Service Bus provides durability and at-least-once delivery.

**Observability**

* No logging in this class. Failures bubble up.
* Relies on Service Bus + central observability (OpenTelemetry middleware).

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Happy path** → enqueue JSON payload, assert SendMessageAsync called.
2. ❌ **Null/empty payload** → ensure exception thrown.
3. 🚨 **Service Bus offline** → exception should propagate.
4. 🛑 **Call DequeueAllAsync** → throws NotSupportedException.

**Code Smells**

* **Low**: No logging on enqueue failures.
* **Info**: No retry/backoff logic (depends on Service Bus SDK).

**Refactoring Suggestions**

* **Add logging around EnqueueAsync** (success/failure).
  + **Effort**: Quick Win
  + **Priority**: 3
* **Consider exposing retry options** (configurable).
  + **Effort**: Small
  + **Priority**: 4

**Contracts & Compatibility**

* Public contract: implements IXeroWebhookQueue.
* Assumes ServiceBus:QueueName config exists.
* Changing queue name/config would break consumers.

**Data Model Notes**

* N/A – messaging, not persistence.

**Confidence**

**High** – full file reviewed, clear Azure Service Bus integration.

**File Metadata**

* **File**: Infrastructure/Webhooks/InMemoryXeroWebhookEventStore.cs
* **Layer/Type**: Infrastructure → Webhooks → In-Memory Store
* **Status**: Reviewed
* **Tokens**: ~1.2k

**ELI5**

This file keeps track of **which Xero webhook events have already been processed** – but only in memory. It’s useful for development and testing, not for production, because the data disappears when the app restarts.

**Purpose and Role**

* Implements IXeroWebhookEventStore.
* Provides **idempotency**: avoids processing the same webhook twice.
* Stores processed events in a thread-safe in-memory dictionary.
* Used in **single-instance, local environments**; production relies on DB-backed repository.

**Detailed Breakdown**

* **Dependencies**:
  + ConcurrentDictionary<Guid, XeroWebhookEvent>.
  + Domain entity: XeroWebhookEvent.
* **Members**:
  + \_processedEvents → key: Guid (event ID), value: event object.
* **Methods**:
  + HasBeenProcessedAsync(Guid eventId) → checks dictionary for key.
  + MarkAsProcessedAsync(XeroWebhookEvent event) → adds/updates dictionary entry.

**Error Handling & Validation**

* No null guards for input event.
* Relies on ConcurrentDictionary for thread safety.
* No persistence → events lost on restart.

**Security Review**

* ✅ No secrets handled.
* ✅ Safe to use in dev/test.
* ⚠️ Should **never** be used in multi-instance production deployments (not distributed).

**Performance & Reliability**

* Fast lookups with ConcurrentDictionary.
* Reliable only for the lifetime of the process.
* No memory eviction policy – potential unbounded growth if left running with many events.

**Observability**

* No logging.
* Cannot audit processed events beyond memory lifetime.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Mark new event** → HasBeenProcessedAsync returns true afterward.
2. ❌ **Check unknown event** → returns false.
3. 🔄 **Re-mark same event** → dictionary overwrites existing entry without issue.
4. 🛑 **Null event** → should throw, but currently would null-reference (test to confirm).
5. 🔄 **Concurrent writes** → validate thread-safety.

**Code Smells**

* **Medium**: No guard against null XeroWebhookEvent.
* **Low**: Memory can grow without bound in long-running processes.

**Refactoring Suggestions**

* **Add null check in MarkAsProcessedAsync**
  + **Effort**: Quick Win
  + **Priority**: 4
* **Add optional TTL / eviction** (to prevent memory growth).
  + **Effort**: Medium
  + **Priority**: 2

**Contracts & Compatibility**

* Contract: implements IXeroWebhookEventStore.
* Compatible with Application layer code expecting HasBeenProcessedAsync and MarkAsProcessedAsync.
* Not durable → must be swapped for DB/Redis implementation in production.

**Data Model Notes**

* N/A – in-memory only, no schema.

**Confidence**

**High** – full file reviewed, straightforward implementation.

**File Metadata**

* **File**: Infrastructure/Webhooks/InMemoryXeroWebhookHistoryRepository.cs
* **Layer/Type**: Infrastructure → Webhooks → In-Memory Repository
* **Status**: Reviewed
* **Tokens**: ~0.8k

**ELI5**

This class stores webhook payloads **only in memory**. It’s meant for development or testing environments, not production, so that webhook messages can be tracked temporarily without using a database.

**Purpose and Role**

* Implements IXeroWebhookHistoryRepository.
* Provides a lightweight, in-memory mechanism for **saving webhook payloads**.
* Useful for local testing, debugging, or running without a backing database.
* Complements InMemoryXeroWebhookEventStore by persisting the actual JSON payload.

**Detailed Breakdown**

* **Dependencies**:
  + ConcurrentBag<string> → stores raw JSON payloads.
  + IXeroWebhookHistoryRepository interface.
* **Members**:
  + \_payloads → thread-safe, append-only collection.
* **Methods**:
  + SaveAsync(string payloadJson, CancellationToken)
    - Adds the JSON string to \_payloads.
    - Returns Task.CompletedTask.

**Error Handling & Validation**

* No null/empty check on payloadJson.
* No exception handling; assumes caller provides valid JSON.
* No eviction → memory grows unbounded.

**Security Review**

* ⚠️ Stores raw webhook payloads → may include **PII or financial data** from Xero.
* ✅ Only used in non-production contexts, but risk if accidentally deployed to production.
* ❌ No redaction or filtering of sensitive fields.

**Performance & Reliability**

* Append-only → no query methods provided.
* Thread-safe via ConcurrentBag.
* Memory may grow without limit under heavy webhook load.

**Observability**

* No logging of saved payloads (good, avoids leaking sensitive data).
* No monitoring of bag size.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Save valid payload** → appears in \_payloads.
2. ❌ **Save null payload** → currently added as null, should be rejected.
3. 🔄 **Save many payloads concurrently** → thread-safety holds.
4. 🛑 **Verify persistence lost on restart** → ensures test-only usage.

**Code Smells**

* **Medium**: Accepts null/empty payloads without validation.
* **Low**: Risk of memory bloat.

**Refactoring Suggestions**

* **Add input validation for payloadJson**.
  + **Effort**: Quick Win
  + **Priority**: 4
* **Add eviction or max payload cap** (to prevent memory overuse).
  + **Effort**: Medium
  + **Priority**: 2

**Contracts & Compatibility**

* Contract: IXeroWebhookHistoryRepository.SaveAsync.
* Only implements **write**, no retrieval.
* Safe for test/dev, but not production.

**Data Model Notes**

* N/A – in-memory only, no database schema.

**Confidence**

**High** – full file reviewed, trivial implementation.

**File Metadata**

* **File**: Infrastructure/Webhooks/InMemoryXeroWebhookQueue.cs
* **Layer/Type**: Infrastructure → Webhooks → In-Memory Queue
* **Status**: Reviewed
* **Tokens**: ~1.2k

**ELI5**

This file creates a **temporary in-memory queue** for Xero webhook events. It allows enqueueing messages and later reading them out, but only while the app is running. It’s a stand-in for Service Bus or a real broker when testing locally.

**Purpose and Role**

* Implements IXeroWebhookQueue.
* Provides a non-persistent message queue for development and testing.
* Wraps a Channel<string> (bounded capacity 1024).
* Allows asynchronous writes (EnqueueAsync) and reads (DequeueAllAsync).

**Detailed Breakdown**

* **Dependencies**:
  + System.Threading.Channels for producer/consumer queue.
  + IXeroWebhookQueue interface.
* **Members**:
  + \_channel → bounded channel, single reader / multiple writers, drop oldest on overflow.
* **Methods**:
  + EnqueueAsync(string eventPayload, CancellationToken)
    - Writes payload to channel.
  + DequeueAllAsync(CancellationToken)
    - Returns an IAsyncEnumerable<string> of all queued payloads.

**Error Handling & Validation**

* No validation of eventPayload (null/empty possible).
* Channel ensures thread-safety.
* Bounded queue avoids unbounded memory growth.
* Overflow policy: drops oldest message silently if queue full.

**Security Review**

* ✅ Only stores payloads in memory.
* ⚠️ Payloads may contain sensitive financial data → ensure not logged when dequeued.
* ⚠️ Should never be used in production (no durability, no distribution).

**Performance & Reliability**

* High throughput, lock-free structure.
* Bounded queue prevents OOM.
* Messages lost if service restarts or channel overflows.

**Observability**

* No logging on enqueue, dequeue, or dropped messages.
* No metrics (e.g., queue depth).

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Enqueue/Dequeue round-trip** → message preserved.
2. ❌ **Null/empty payload** → should reject, currently not validated.
3. 🔄 **Queue overflow** → oldest message dropped, test correctness.
4. 🛑 **Concurrent producers and consumer** → thread-safety confirmed.
5. ⏳ **Cancellation token** → verify it cancels properly.

**Code Smells**

* **Medium**: No validation on payload.
* **Medium**: Silent message loss on overflow – may surprise consumers.
* **Info**: No visibility into queue size or dropped messages.

**Refactoring Suggestions**

* **Validate payload input** before enqueue.
  + **Effort**: Quick Win
  + **Priority**: 4
* **Log dropped messages (overflow)**.
  + **Effort**: Small
  + **Priority**: 3
* **Expose queue depth metric** for observability.
  + **Effort**: Medium
  + **Priority**: 2

**Contracts & Compatibility**

* Contract: Implements IXeroWebhookQueue.
* Compatible with same consumers as AzureServiceBusWebhookQueue.
* Not interchangeable in production – dev/test only.

**Data Model Notes**

* N/A – in-memory only.

**Confidence**

**High** – full file reviewed, straightforward channel-based implementation.

**File Metadata**

* **File**: Infrastructure/Webhooks/MessageBrokerXeroWebhookQueue.cs
* **Layer/Type**: Infrastructure → Webhooks → Queue Stub
* **Status**: Reviewed
* **Tokens**: ~1.7k

**ELI5**

This class is a **placeholder implementation** of the webhook queue that logs the event payload instead of actually sending it to a real message broker. It’s meant for scaffolding and local use until a proper integration with a broker (RabbitMQ, Kafka, etc.) is built.

**Purpose and Role**

* Implements IXeroWebhookQueue.
* Logs webhook payloads for visibility.
* Does **not** provide real queuing or persistence.
* Useful for development, not production.

**Detailed Breakdown**

* **Dependencies**:
  + ILogger for logging.
  + IXeroWebhookQueue interface.
* **Members**:
  + \_logger → logs payload activity.
* **Methods**:
  + EnqueueAsync(string eventPayload, CancellationToken)
    - Logs payload length.
    - Returns Task.CompletedTask.
  + DequeueAllAsync(CancellationToken)
    - Not implemented; logs comment suggests a real broker should supply this.
    - Currently just await Task.CompletedTask; yield break;.

**Error Handling & Validation**

* No validation for eventPayload.
* No exception handling – relies on logging only.

**Security Review**

* ⚠️ Payload content may include PII or financial details → currently logs length only, which is safe.
* ✅ Does not log full payload (good).
* ⚠️ Should not be deployed in production (no durability, no delivery guarantees).

**Performance & Reliability**

* Lightweight, no storage.
* Reliable only for logging visibility, not event processing.
* No retries, no queue semantics.

**Observability**

* ✅ Logs every enqueue with payload length.
* ❌ No logging for dequeue (stubbed).
* ❌ No metrics.

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Enqueue valid payload** → log entry written.
2. ❌ **Enqueue null/empty payload** → still logs, should be validated.
3. 🔄 **Call DequeueAllAsync** → currently yields nothing.
4. 🛑 **Simulate high volume** → confirm logging doesn’t cause bottleneck.

**Code Smells**

* **High**: Stub, not a functional implementation.
* **Medium**: No validation for payload.
* **Info**: Dequeue method misleadingly implemented (but empty).

**Refactoring Suggestions**

* **Implement a real broker-backed version** (RabbitMQ, Kafka, etc.).
  + **Effort**: Large
  + **Priority**: 5
* **Add validation for payload input**.
  + **Effort**: Quick Win
  + **Priority**: 3
* **Make DequeueAllAsync explicitly throw NotSupportedException** instead of yielding silently.
  + **Effort**: Small
  + **Priority**: 4

**Contracts & Compatibility**

* Implements IXeroWebhookQueue, but doesn’t actually fulfill the contract fully.
* Safe for dev/test scaffolding only.
* Replacing with real broker implementation requires no contract change.

**Data Model Notes**

* N/A – no persistence.

**Confidence**

**High** – full file reviewed, intentionally stubbed.

**File Metadata**

* **File**: Infrastructure/Webhooks/XeroWebhookWorker.cs
* **Layer/Type**: Infrastructure → Webhooks → Background Worker
* **Status**: Reviewed
* **Tokens**: ~4.0k

**ELI5**

This worker listens to an **Azure Service Bus queue of Xero webhook messages**. When a new event arrives, it validates it, checks if it has already been processed, and if not, records it and sends a MediatR command so the rest of the system can handle it.

**Purpose and Role**

* Runs continuously as a background service (BackgroundService).
* Reads messages from an Azure Service Bus queue (xero-webhooks).
* Ensures **idempotency** (skips events already processed).
* Sends valid webhook events into the Application layer via MediatR.
* Stores processing history in IXeroWebhookEventStore.

**Detailed Breakdown**

* **Dependencies**:
  + ServiceBusClient, ServiceBusProcessor (Azure SDK).
  + IScopeFactory (for DI scope per message).
  + IMediator (MediatR) to publish XeroWebhookReceivedCommand.
  + IXeroWebhookEventStore (for idempotency).
  + ILogger<XeroWebhookWorker> for observability.
* **Key Workflow**:
  + ExecuteAsync registers handlers and starts Service Bus message pump.
  + On message (ProcessMessageAsync):
    - Reads body as string.
    - Logs receipt.
    - Parses JSON → extracts eventId.
    - Validates eventId exists.
    - Checks store if already processed.
    - If new → publishes XeroWebhookReceivedCommand to MediatR.
    - Records event in IXeroWebhookEventStore.
    - Marks message complete.
  + On error (ErrorHandler): logs exception.
* **Domain coupling**:
  + Uses domain entity XeroWebhookEvent.
  + Stores ProcessedAtUtc.

**Error Handling & Validation**

* Logs and dead-letters messages if eventId is missing.
* Logs and skips if event already processed.
* Catches and logs exceptions in ErrorHandler.
* No retry/backoff strategy (relies on Service Bus retry policy).

**Security Review**

* ✅ Payload never logged in full, only message length.
* ✅ Tokenized via Service Bus, no raw credentials in code.
* ⚠️ Webhook payload may include PII/financial data; must not leak to logs.
* ⚠️ Ensure Service Bus connection string is in **Key Vault / env vars**.

**Performance & Reliability**

* Asynchronous message processing.
* One event processed at a time by default processor (can scale concurrency).
* Idempotency prevents duplicate processing.
* Relies on Service Bus for retries & DLQ.
* Potential performance bottleneck if many webhook events processed sequentially.

**Observability**

* Logs at **Info**: when message received, when skipped, when completed.
* Logs at **Warning**: invalid payload (missing eventId).
* Logs at **Error**: exceptions.
* Good coverage, but lacks metrics (processed count, error count).

**Testability & Coverage**

**Suggested test cases**:

1. ✅ **Valid new event** → processed, MediatR command sent.
2. 🔄 **Duplicate eventId** → skipped, not processed again.
3. ❌ **Missing eventId in payload** → logs warning, dead-letters message.
4. 🚨 **Message handling throws exception** → logged, Service Bus should retry.
5. ⏳ **Graceful shutdown** → cancels ExecuteAsync and stops processor.
6. 📊 **High volume** → validate throughput and idempotency.

**Code Smells**

* **Medium**: No explicit retry/backoff beyond Service Bus defaults.
* **Low**: Parsing assumes JSON structure; fragile if Xero changes schema.
* **Info**: EventId validation uses TryGetProperty – safe, but could extract tenant info too.

**Refactoring Suggestions**

* **Add metrics collection** (events processed, skipped, errored).
  + **Effort**: Small
  + **Priority**: 3
* **Make retry/backoff configurable** (currently hidden in Service Bus defaults).
  + **Effort**: Medium
  + **Priority**: 4
* **Extend validation** (check tenantId, category, type).
  + **Effort**: Small
  + **Priority**: 2

**Contracts & Compatibility**

* Contract: processes messages from Service Bus queue (xero-webhooks).
* Consumes Xero webhook schema JSON.
* Produces Application command XeroWebhookReceivedCommand.
* Backward compatibility risk if Xero changes webhook schema.

**Data Model Notes**

* XeroWebhookEvent stored with EventId, ProcessedAtUtc.
* Idempotency depends on DB or memory store implementation.

**Confidence**

**High** – full file reviewed, clear orchestration of Service Bus → MediatR workflow.