**File Review**

**File:** Tests/Tests.csproj  
**Layer/Type:** Tests – Project Configuration  
**Status:** Reviewed  
**Tokens (approx.):** ~180

**🧾 ELI5**

This file is the blueprint that tells .NET how to build and run the test suite. It declares that the tests target .NET 8, pulls in libraries like xUnit, Moq, and FluentAssertions, and ensures the tests can reference RoadmApp’s Domain, Application, Infrastructure, and Web projects.

**🎯 Purpose and Role**

* Defines the **test project** for RoadmApp.
* Ensures tests are built against **.NET 8.0** using the Web SDK (to match the runtime environment).
* Adds references to **all major RoadmApp projects**, enabling unit, integration, and end-to-end testing.
* Declares NuGet dependencies on **testing frameworks** and **mocking tools**.
* Copies configuration (appsettings.json) to the output so tests can run with settings.

**🔍 Detailed Breakdown**

**Framework & SDK:**

* Uses Microsoft.NET.Sdk.Web.
* Targets net8.0.

**Test Libraries:**

* xunit, xunit.runner.visualstudio → test execution.
* FluentAssertions → expressive assertions.
* FluentValidation → allows validation in tests.
* Moq → mocking framework for dependencies.
* Microsoft.NET.Test.Sdk → required test runner SDK.

**Other Dependencies:**

* coverlet.collector & coverlet.msbuild → code coverage support.

**Project References:**

* Application/Application.csproj
* Domain/Domain.csproj
* Infrastructure/Infrastructure.csproj
* Web/Web.csproj

**Build Behavior:**

* appsettings.json is copied to test output (PreserveNewest).
* Includes Infrastructure\Web folder → integration/e2e tests may rely on full runtime environment.

**⚠️ Error Handling & Validation**

* No error handling (as expected for project file).
* Validation of test inputs occurs in test code, not here.

**🔐 Security Review**

* ✅ No secrets hard-coded.
* ⚠️ appsettings.json is copied to test output. Need to confirm **test-specific safe values** are used, not production connection strings or OAuth credentials.
* Should rely on Key Vault or mock providers in tests (not real secrets).

**⚡ Performance & Reliability**

* References both unit-test libraries and full Web project → CI/CD runs may be **slower** if integration tests execute alongside unit tests.
* Code coverage (coverlet) increases reliability of test metrics.
* Dependency footprint is appropriate; no heavy frameworks.

**📊 Observability**

* No logging setup here.
* Observability is expected in test execution (assert logs, metrics from Application layer).

**🧪 Testability & Coverage**

This configuration enables:

* **Unit Tests** → Application & Domain logic.
* **Integration Tests** → Infrastructure (Dapper, Redis, Key Vault).
* **End-to-End Tests** → Web controllers and API flows.

**Suggested test categories to verify coverage:**

* **Positive:** Valid user login via Web API → expect success.
* **Negative:** Attempt password reset with invalid token → expect failure.
* **Edge:** Empty dataset returned from Dapper query → should not throw.
* **Integration:** OAuth token refresh workflow → confirm rotation rules.
* **Resilience:** Redis unavailable → fallback to in-memory cache.

**🧹 Code Smells**

* **Info:** Project references Web directly → risks mixing slow integration/e2e tests into the same suite as unit tests. Could make local dev runs heavy.
* **Info:** Copying appsettings.json → risk of using unsafe config if not explicitly test-scoped.

**🔧 Refactoring Suggestions**

1. **Split tests by category** → create separate projects:
   * Tests.Unit (Domain/Application only).
   * Tests.Integration (Infrastructure/Web).
   * Effort: **Medium**, Priority: **3**.
2. **Ensure safe config isolation** → use appsettings.Test.json with dummy secrets.
   * Effort: **Small**, Priority: **4**.
3. **Introduce test fixtures** for expensive dependencies (Redis, DB connections) to avoid re-initialisation in every test.
   * Effort: **Small**, Priority: **3**.

**📜 Contracts & Compatibility**

* Uses standard .NET 8 test SDK and libraries → stable and forward-compatible.
* Any removal of project references (e.g., to Web) could break integration tests.
* Libraries like xUnit, Moq, FluentAssertions are widely supported in CI/CD pipelines.

**🗄️ Data Model Notes**

* Not applicable (project config, no schema).

**✅ Confidence**

**High** – Full file was available; analysis cross-checked with RoadmApp ADRs and review guidelines.

**File Review**

**File:** Tests/Application/Features/Ingestion/Queries/GetTenantLogsQueryHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Query handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~200

**🧾 ELI5**

This test checks that when the system asks for logs related to a specific tenant, the query handler correctly calls the log service and returns the expected log entries.

**🎯 Purpose and Role**

* Validates **Application layer logic** for GetTenantLogsQueryHandler.
* Ensures the query handler integrates correctly with ICallLogService.
* Verifies correct handling of a **query object** (GetTenantLogsQuery) and returns a success response with logs.
* Helps guarantee that tenant log retrieval works as expected in the RoadmApp ingestion pipeline.

**🔍 Detailed Breakdown**

* **Testing framework:** Uses xUnit ([Fact]) with Moq.
* **Mocks:** ICallLogService is mocked.
* **Setup:**
  + Configures GetTenantLogsAsync to return a List<ApiCallLogEntry>.
  + Each log entry includes CallTime, Endpoint, StatusCode, and Success.
* **Handler under test:** GetTenantLogsQueryHandler.
* **Execution:**
  + Calls handler.Handle(new GetTenantLogsQuery(...)).
* **Assertions:**
  + result.IsSuccess is true.
  + result.Value.Logs is populated (not empty).

**⚠️ Error Handling & Validation**

* Only the **happy path** is tested (logs returned).
* No coverage for:
  + Empty log list.
  + Null/exception from ICallLogService.
  + Invalid tenant IDs.
  + Cancellation token usage.

**🔐 Security Review**

* ✅ No secrets or PII.
* Logs tested here are mock entries; real logs may contain sensitive data (ensure sanitisation at Application layer, not in test).
* No risk of leaking OAuth tokens or passwords.

**⚡ Performance & Reliability**

* Test is **lightweight** (in-memory mocks).
* Executes async methods correctly with await.
* No parallelism/concurrency issues.

**📊 Observability**

* No explicit logging tested.
* Focus is purely on data returned by handler.
* Suggestion: Include test ensuring correlation IDs or trace context are passed along if implemented in handler.

**🧪 Testability & Coverage**

**Covered:**

* Positive happy path with valid logs.

**Missing (should add):**

1. **Negative:** ICallLogService returns empty list → result should still be success but logs empty.
2. **Negative:** ICallLogService throws exception → handler should wrap in failure response.
3. **Edge:** Tenant ID exists but no logs within timeframe.
4. **Edge:** Cancellation token is cancelled mid-call.
5. **Boundary:** Very large log list returned → handler should return efficiently without truncation.

**🧹 Code Smells**

* **Medium:** Only one test case → low coverage.
* **Low:** Uses inline object creation for logs → may duplicate setup in future tests (extract fixture?).
* **Info:** Test names (ReturnsLogsForTenant) are descriptive but could follow **Given/When/Then** convention for clarity.

**🔧 Refactoring Suggestions**

1. Add **test fixture/factory** for common log entries.
   * Effort: Small, Priority: 2.
2. Expand coverage to include **negative and edge cases** listed above.
   * Effort: Medium, Priority: 4.
3. Rename test method to follow **BDD naming** (Should\_ReturnLogs\_When\_TenantHasLogs).
   * Effort: Quick Win, Priority: 3.

**📜 Contracts & Compatibility**

* Relies on GetTenantLogsQuery and ICallLogService.
* Contract risk: if ICallLogService.GetTenantLogsAsync changes signature, these tests must be updated.
* DTOs used (ApiCallLogEntry) are stable Application-layer contracts.

**🗄️ Data Model Notes**

* Not applicable (test-only, mocks logs rather than persisting).

**✅ Confidence**

**High** – Entire file available, clear purpose, mapped correctly to Application feature under test.

**File Review**

**File:** Tests/Application/Features/Organisation/Queries/GetOrganisationInfoQueryHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Query handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~50 (very small file)

**🧾 ELI5**

This file is supposed to test the query handler that retrieves organisation information, but it currently contains **only a header and metadata comments** — no actual test implementation.

**🎯 Purpose and Role**

* Intended to validate the Application feature GetOrganisationInfoQueryHandler.
* Should ensure that organisation details (like ID, name, creation date) are returned correctly.
* Would normally test integration with persistence via Dapper, but using mocks in the Application layer.
* Currently **unimplemented** → placeholder only.

**🔍 Detailed Breakdown**

* Contains only file header with SPDX license info and metadata (created: 2025-06-16).
* No namespaces, classes, or test methods defined.
* No references to xUnit, Moq, or query handler code.

**⚠️ Error Handling & Validation**

* None present — file has no executable code.

**🔐 Security Review**

* ✅ No secrets.
* ✅ No risk of logging sensitive values.

**⚡ Performance & Reliability**

* N/A — no test logic.

**📊 Observability**

* N/A — no test logic.

**🧪 Testability & Coverage**

**Current coverage:** 0% for GetOrganisationInfoQueryHandler.

**Recommended test cases:**

1. **Positive:** Valid organisation ID → handler returns correct organisation info.
2. **Negative:** Invalid organisation ID (non-existent) → handler returns failure.
3. **Edge:** Null/empty organisation ID → handler should reject.
4. **Edge:** Organisation exists but has minimal data (empty name) → handler handles gracefully.
5. **Integration:** Query executed against test DB with seeded organisation.

**🧹 Code Smells**

* **Critical:** Empty placeholder test file — no assertions, no coverage.

**🔧 Refactoring Suggestions**

1. Implement unit tests for GetOrganisationInfoQueryHandler:
   * Mock repository/service to return organisation data.
   * Assert handler wraps result in success response.
   * Effort: **Small**, Priority: **5** (critical for coverage).
2. Add failure path tests (missing/invalid IDs).
   * Effort: **Small**, Priority: **4**.
3. Consider integration test variant with seeded DB record (optional).
   * Effort: **Medium**, Priority: **3**.

**📜 Contracts & Compatibility**

* Will depend on GetOrganisationInfoQuery and IOrganisationService/repository.
* Changes to query contract will require updating tests.

**🗄️ Data Model Notes**

* Should map to the **Organisation** entity (Domain).
* DB-backed tests should assert constraints: OrganisationId PK, Name not null.

**✅ Confidence**

**High** – Full file was available; confirmed it is only a stub with no tests.

**File Review**

**File:** Tests/Application/Features/Polling/Commands/SavePollingScheduleCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~450

**🧾 ELI5**

This file tests the command handler that saves polling schedules. It checks that when a valid polling schedule command is sent, the handler calls the service correctly and returns a success response.

**🎯 Purpose and Role**

* Validates **Application layer command handler** SavePollingScheduleCommandHandler.
* Ensures correct interaction with IPollingSettingsService.
* Confirms that polling schedules can be persisted through the handler.
* Protects against regression in the core **background job scheduling** feature of RoadmApp.

**🔍 Detailed Breakdown**

* **Frameworks:** Uses xUnit ([Fact]), Moq, and FluentAssertions.
* **Mocks:** IPollingSettingsService.
* **Commands tested:** SavePollingScheduleCommand with different frequencies (Daily, Weekly, Hourly, None).
* **Test cases:**
  1. **Should\_Succeed\_For\_Valid\_Command** → sets up mock service to return CompletedTask.
     + Asserts result is IsSuccess = true.
  2. **Passes\_Frequency\_To\_Service** (parameterised with [InlineData]):
     + Tests that command passes frequency correctly to the service.
     + Covers Hourly, Daily, Weekly, None.
     + Verifies UpsertAsync was called with correct arguments.

**⚠️ Error Handling & Validation**

* Only **happy path** tested.
* Missing scenarios:
  + Service throws exception → should result in failure.
  + Invalid frequency outside enum values.
  + Null cancellation token.
  + Conflicts with existing schedules.

**🔐 Security Review**

* ✅ No secrets.
* ✅ No PII.
* Polling schedules are non-sensitive, safe to log.

**⚡ Performance & Reliability**

* Tests are **fast** (in-memory mocks).
* Correctly async/awaited.
* Parameterised tests ensure multiple cases validated efficiently.

**📊 Observability**

* No observability tested (no logging assertions).
* Handler may include logging; tests could verify logs for traceability.

**🧪 Testability & Coverage**

**Covered:**

* Valid command results in success.
* Correct frequency passed to service.

**Missing:**

1. **Negative:** Service throws exception → handler should return failure.
2. **Edge:** Frequency not defined in enum → should reject.
3. **Edge:** Command with null values (organisation ID or frequency).
4. **Concurrency:** Multiple save commands for same org → ensure only one schedule persists.
5. **Integration:** Run with in-memory DB/service to ensure persistence round-trip.

**🧹 Code Smells**

* **Medium:** No failure case tests → risk of false sense of coverage.
* **Low:** Inline data only covers valid enum values → does not test invalid cases.
* **Info:** Test names use mixed conventions (Should\_Succeed\_For\_Valid\_Command vs Passes\_Frequency\_To\_Service).

**🔧 Refactoring Suggestions**

1. Add failure path tests (service throws exception).
   * Effort: Small, Priority: 5.
2. Add guard clause tests for invalid frequencies.
   * Effort: Small, Priority: 4.
3. Unify test naming to **Given\_When\_Then** format.
   * Effort: Quick Win, Priority: 2.
4. Extract a **command builder/helper** to reduce duplication.
   * Effort: Small, Priority: 3.

**📜 Contracts & Compatibility**

* Depends on SavePollingScheduleCommand and IPollingSettingsService.UpsertAsync.
* If UpsertAsync signature changes (e.g., new parameters), tests must be updated.
* Polling frequency enum is part of domain contract; adding/removing values requires test updates.

**🗄️ Data Model Notes**

* Indirectly relates to PollingSetting entity in Domain.
* Tests verify command → service interaction, not persistence.
* DB schema for PollingSettings (OrgId, Schedule, RunTime) must align with tested behaviour.

**✅ Confidence**

**High** – Full file reviewed, clear coverage of handler happy path, but gaps remain for negative/edge cases.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/AssignRoleCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~450

**🧾 ELI5**

This file tests the handler responsible for assigning roles to users in organisations. It checks both successful and failed role assignment scenarios, ensuring the service is called correctly and the result is handled.

**🎯 Purpose and Role**

* Validates **Application layer command handler** AssignRoleCommandHandler.
* Ensures it interacts properly with IAccessService (the service that manages role assignments).
* Confirms correct handling of success and failure responses.
* Protects against regressions in **authorisation role management** workflows.

**🔍 Detailed Breakdown**

* **Frameworks:** Uses xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IAccessService (service under test).
  + ILogger<AssignRoleCommandHandler> (logger mock).
* **Handler under test:** AssignRoleCommandHandler.
* **Test cases:**
  + **Should\_Succeed\_When\_Service\_Succeeds**
    - Creates a command (AssignRoleCommand) with UserId, OrganisationId, RoleId.
    - Mocks IAccessService.AssignRoleAsync to return Result<bool>.Success(true).
    - Asserts: IsSuccess = true, no error.
  + **Should\_Fail\_When\_Service\_Fails**
    - Mocks service to return Result<bool>.Failure("Assignment failed.").
    - Asserts: IsSuccess = false, error contains "failed".

**⚠️ Error Handling & Validation**

* Good coverage of both success and failure cases.
* Missing tests for:
  + Null or invalid UserId, OrganisationId, RoleId.
  + Service throwing exception (not just returning failure).
  + Cancellation token behaviour.

**🔐 Security Review**

* ✅ No secrets or PII in tests.
* ⚠️ Role assignment is **security-critical**: need to ensure handler prevents duplicate or invalid assignments (not tested here).
* Logger is mocked, so no risk of sensitive logging in tests.

**⚡ Performance & Reliability**

* Lightweight, no external dependencies.
* Async/await handled properly.
* Only two test cases → limited validation of reliability across multiple role assignment scenarios.

**📊 Observability**

* Logger is mocked but **not verified**.
* No assertions that log messages are written (missed opportunity for observability testing).

**🧪 Testability & Coverage**

**Covered:**

* Happy path (service succeeds).
* Failure path (service returns failure).

**Missing (should add):**

1. **Negative:** Service throws exception → handler should wrap in failure response.
2. **Edge:** Command with Guid.Empty values → should fail validation.
3. **Edge:** Multiple assignments to same user/org/role → service should prevent duplicates.
4. **Observability:** Verify logger writes appropriate messages on success/failure.

**🧹 Code Smells**

* **Medium:** Only two test cases → limited coverage.
* **Low:** Logger is injected but not verified.
* **Info:** Test names use Should\_\* but inconsistent detail level.

**🔧 Refactoring Suggestions**

1. Add **exception handling tests** for robustness.
   * Effort: Small, Priority: 5.
2. Add **validation tests** for Guid.Empty inputs.
   * Effort: Small, Priority: 4.
3. Add **logger verification** (e.g., logger.LogError on failure).
   * Effort: Small, Priority: 3.
4. Expand coverage to include **duplicate role assignments**.
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* Depends on AssignRoleCommand and IAccessService.AssignRoleAsync.
* Any signature change in service or command will break these tests.
* DTOs used (AssignRoleCommand) are part of Application layer contract → must remain stable.

**🗄️ Data Model Notes**

* Indirectly linked to UserOrgRole entity in Domain.
* DB schema must enforce uniqueness on (UserId, OrganisationId, RoleId).

**✅ Confidence**

**High** – Full file reviewed, clear structure, but missing edge/failure cases.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/AssignUsersCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~250

**🧾 ELI5**

This file tests the command handler that assigns users to an organisation. It verifies that when a valid command is processed, the handler calls the organisation service and succeeds.

**🎯 Purpose and Role**

* Validates **Application layer command handler** AssignUsersCommandHandler.
* Ensures the handler correctly interacts with IOrganisationService.
* Confirms that user assignment to organisations is processed successfully.
* Helps guarantee that RoadmApp’s **multi-tenant user assignment logic** works as intended.

**🔍 Detailed Breakdown**

* **Frameworks:** Uses xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IOrganisationService (service under test).
  + ILogger<AssignUsersCommandHandler> injected but not explicitly verified.
* **Handler under test:** AssignUsersCommandHandler.
* **Test case:**
  + Creates an AssignUsersCommand with:
    - OrganisationId (new Guid).
    - UserIds (list of Guids).
  + Mocks AssignUsersAsync to return Result<bool>.Success(true).
  + Asserts result is IsSuccess = true, error is null.

**⚠️ Error Handling & Validation**

* Only tests the **happy path** (service returns success).
* Missing scenarios:
  + Service returns failure.
  + Service throws exception.
  + Command contains empty/invalid user IDs or organisation ID.
  + Duplicate users assigned to the same organisation.

**🔐 Security Review**

* ✅ No secrets or PII in test.
* ⚠️ Multi-tenant security risk: duplicate or incorrect assignments could cause privilege escalation. This test does not validate negative paths.

**⚡ Performance & Reliability**

* Lightweight, no external dependencies.
* Async/await handled correctly.
* No concurrency or bulk assignment edge cases tested (e.g., very large user list).

**📊 Observability**

* Logger injected but not used in assertions.
* No verification of log outputs (missed observability opportunity).

**🧪 Testability & Coverage**

**Covered:**

* Happy path where service returns success.

**Missing (should add):**

1. **Failure path:** Service returns failure → handler should return failure result.
2. **Exception path:** Service throws exception → handler should handle gracefully.
3. **Validation:** Command with empty UserIds should fail.
4. **Edge:** Assign same user twice → ensure deduplication.
5. **Observability:** Verify logging on success/failure.

**🧹 Code Smells**

* **Medium:** Only one test case → extremely limited coverage.
* **Low:** Logger dependency mocked but unused.
* **Info:** Test naming is descriptive but doesn’t follow consistent BDD style.

**🔧 Refactoring Suggestions**

1. Add negative and exception handling tests.
   * Effort: Small, Priority: 5.
2. Add validation edge-case tests for empty/duplicate IDs.
   * Effort: Small, Priority: 4.
3. Verify logger interactions on success/failure.
   * Effort: Small, Priority: 3.
4. Consider parameterised tests for multiple user assignment scenarios.
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* Depends on AssignUsersCommand and IOrganisationService.AssignUsersAsync.
* Any change in command structure (e.g., switching UserIds type) will break test.
* Part of Application layer contract → must remain stable for multi-tenant logic.

**🗄️ Data Model Notes**

* Indirectly tied to UserOrgRole join table in Domain.
* DB schema must enforce uniqueness (UserId, OrganisationId) to prevent duplicates.

**✅ Confidence**

**High** – File fully available, clear intent, but test coverage is minimal.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/ChangePasswordCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~500

**🧾 ELI5**

This file tests the handler responsible for changing user passwords. It checks that a valid current password allows a new password to be set, and that invalid credentials correctly prevent the change.

**🎯 Purpose and Role**

* Validates **Application layer command handler** ChangePasswordCommandHandler.
* Ensures integration with IUserService for authentication and password updates.
* Confirms that the workflow only succeeds if the current password is valid.
* Helps protect RoadmApp’s **account security model** by testing password lifecycle logic.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IUserService → handles authentication and password updates.
  + ILogger<ChangePasswordCommandHandler>.
* **Handler under test:** ChangePasswordCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Succeed\_When\_Current\_Password\_Is\_Valid\_And\_Change\_Succeeds**
    - Creates command with UserId, CurrentPassword, NewPassword.
    - Mocks AuthenticateAsync → returns Success(RoadmappUser).
    - Mocks ChangePasswordAsync → returns Success(true).
    - Asserts: IsSuccess = true, error is null.
  + **Handle\_Should\_Fail\_When\_Current\_Password\_Is\_Invalid**
    - Mocks AuthenticateAsync → returns Failure("Invalid username or password.").
    - Executes handler with invalid current password.
    - Asserts: IsSuccess = false, error contains "incorrect".

**⚠️ Error Handling & Validation**

* ✅ Valid and invalid password scenarios are tested.
* ❌ Missing tests for:
  + New password fails validation (too short, too weak).
  + Service throws exception.
  + Password change request for deactivated user.
  + Null/empty NewPassword input.

**🔐 Security Review**

* ✅ No secrets or PII leaked in tests.
* ⚠️ Passwords are provided in plaintext test data ("OldPassword123!", "NewPassword456!"). This is acceptable in unit tests but should be reviewed carefully to avoid logs or leaks.
* Ensures authentication is verified before allowing password change.
* ❌ No coverage for security policies like password complexity or history (must be enforced in Application/Infrastructure).

**⚡ Performance & Reliability**

* Lightweight, runs quickly with mocks.
* Async/await handled correctly.
* No concurrency or brute force scenarios tested.

**📊 Observability**

* Logger injected but not verified.
* No assertions to check if success/failure events are logged.

**🧪 Testability & Coverage**

**Covered:**

* Valid password change.
* Invalid current password.

**Missing (should add):**

1. **Failure path:** ChangePasswordAsync returns failure (e.g., DB error).
2. **Validation:** New password is empty or weak → handler rejects.
3. **Exception path:** IUserService throws exception.
4. **Edge:** User disabled → password change should fail.
5. **Observability:** Verify logs written on success/failure.

**🧹 Code Smells**

* **Medium:** Only two scenarios tested → coverage too narrow.
* **Low:** Logger is unused in assertions.
* **Info:** Test naming is consistent but could follow Given\_When\_Then convention.

**🔧 Refactoring Suggestions**

1. Add test for **failed password update** (service failure).
   * Effort: Small, Priority: 5.
2. Add **validation tests** for weak/empty passwords.
   * Effort: Small, Priority: 4.
3. Verify **logging interactions**.
   * Effort: Small, Priority: 3.
4. Parameterise tests to cover **multiple invalid password scenarios**.
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* Relies on ChangePasswordCommand and IUserService.AuthenticateAsync + ChangePasswordAsync.
* Contract risk: if password validation rules change, these tests must be updated.
* Ensures user account contract is consistent with domain security requirements.

**🗄️ Data Model Notes**

* Tied to RoadmappUser entity (Domain).
* DB schema must enforce password hash not null.
* Password hashing/peppering not tested here (Infrastructure concern).

**✅ Confidence**

**High** – Full file available; solid base tests but limited coverage.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/CreateAdminUserCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~600

**🧾 ELI5**

This file tests the handler that creates an admin user account. It verifies that a new admin user is created when valid details are supplied, and that the process fails gracefully if the username or email already exists.

**🎯 Purpose and Role**

* Validates **Application layer command handler** CreateAdminUserCommandHandler.
* Ensures correct integration with IUserService.
* Confirms success/failure scenarios for creating new admin users.
* Protects RoadmApp’s **administration onboarding workflow** from regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IUserService → provides admin creation logic.
* **Handler under test:** CreateAdminUserCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Return\_Success\_Response\_When\_Admin\_User\_Is\_Created**
    - Creates CreateAdminUserCommand with Email, Username, FullName.
    - Mocks InviteAdminAsync to return Success(new RoadmappUser) with details.
    - Asserts:
      * UserId matches.
      * Email, Username, FullName match input.
      * Message = "Admin user created successfully."
      * RequiresPasswordSetup = true.
      * RequiresEmailVerification = false.
  + **Handle\_Should\_Return\_Error\_Message\_When\_UserService\_Fails**
    - Creates CreateAdminUserCommand with existing email/username.
    - Mocks InviteAdminAsync to return Failure("Admin with this username or email already exists.").
    - Asserts:
      * Message contains failure reason.
      * Email, Username preserved from request.
      * RequiresPasswordSetup = true.
      * RequiresEmailVerification = false.

**⚠️ Error Handling & Validation**

* ✅ Covers success and duplicate/failure scenarios.
* ❌ Missing tests for:
  + Null/empty email or username.
  + Invalid email format.
  + Failure due to infrastructure (DB down, email service unavailable).
  + Edge case: creating admin with pre-existing but inactive account.

**🔐 Security Review**

* ✅ No secrets or PII leaked.
* ✅ Ensures unique email/username enforced.
* ❌ Does not test password policies (handled separately).
* ❌ No verification that admin users are restricted to minimal required permissions at creation.

**⚡ Performance & Reliability**

* Tests are lightweight (mock-based).
* Async/await handled correctly.
* No performance bottlenecks tested (e.g., concurrent admin creations).

**📊 Observability**

* Logger not used in this handler/test.
* No assertions for logging or audit events.

**🧪 Testability & Coverage**

**Covered:**

* Successful creation of admin user.
* Failure when username/email already exists.

**Missing (should add):**

1. **Validation:** Empty or invalid email → reject.
2. **Failure path:** Infrastructure/service throws exception → handler should fail gracefully.
3. **Edge:** Create admin with inactive status.
4. **Security:** Attempt to assign additional roles beyond admin.
5. **Observability:** Verify log entries or audit trail.

**🧹 Code Smells**

* **Medium:** Only two test cases → insufficient coverage.
* **Low:** Email validation not covered.
* **Info:** RequiresPasswordSetup and RequiresEmailVerification hardcoded expectations → might become fragile if logic changes.

**🔧 Refactoring Suggestions**

1. Add **validation tests** for email and username.
   * Effort: Small, Priority: 5.
2. Add **exception handling test** (service throws).
   * Effort: Small, Priority: 4.
3. Verify **logging/audit** to confirm admin creation is tracked.
   * Effort: Small, Priority: 3.
4. Add concurrency simulation for duplicate admin requests.
   * Effort: Medium, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on CreateAdminUserCommand and IUserService.InviteAdminAsync.
* Any contract changes (fields added to command, return DTO expanded) will break tests.
* Strong dependency on domain entity RoadmappUser.

**🗄️ Data Model Notes**

* Admin users map to RoadmappUser entity with elevated role.
* DB schema must enforce unique constraints on Username and Email.
* Must ensure audit trail of admin creation (outside scope of this test).

**✅ Confidence**

**High** – File fully available; good coverage of primary paths but missing validation, edge, and observability scenarios.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/EditOrganisationCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~300

**🧾 ELI5**

This file tests the handler that edits organisation details. It checks that when a valid organisation ID and new name are provided, the update succeeds.

**🎯 Purpose and Role**

* Validates **Application layer command handler** EditOrganisationCommandHandler.
* Ensures correct integration with IOrganisationService.
* Confirms that organisation updates return a successful response.
* Guards RoadmApp’s **multi-tenant organisation management** from regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IOrganisationService.
  + ILogger<EditOrganisationCommandHandler> (created via Moq.Of<>).
* **Handler under test:** EditOrganisationCommandHandler.
* **Test case:**
  + Creates EditOrganisationCommand with OrganisationId and OrganisationName = "Edited Name".
  + Mocks EditOrganisationAsync → returns Success(true).
  + Asserts:
    - IsSuccess = true.
    - Error = null.

**⚠️ Error Handling & Validation**

* ✅ Happy path (edit success) tested.
* ❌ Missing scenarios:
  + Service returns failure (e.g., org not found).
  + Service throws exception.
  + Invalid input: empty or null organisation name.
  + Editing a deactivated/deleted organisation.

**🔐 Security Review**

* ✅ No secrets or PII in test.
* ⚠️ Organisation names may contain PII in production (customer names). This test doesn’t check for sanitisation.

**⚡ Performance & Reliability**

* Lightweight, runs quickly.
* Async/await handled correctly.
* No concurrency or large-batch editing scenarios tested.

**📊 Observability**

* Logger is injected but unused.
* No verification of logs for auditability (important for organisation changes).

**🧪 Testability & Coverage**

**Covered:**

* Success path for editing organisation.

**Missing (should add):**

1. **Failure path:** Organisation not found → service returns failure.
2. **Exception path:** Service throws exception.
3. **Validation:** Empty/invalid organisation name rejected.
4. **Edge:** Attempt to edit organisation with Guid.Empty.
5. **Observability:** Verify log entries when update succeeds/fails.

**🧹 Code Smells**

* **Medium:** Only one test case → extremely limited coverage.
* **Low:** Logger unused.
* **Info:** Test name doesn’t follow BDD convention (Should\_\*).

**🔧 Refactoring Suggestions**

1. Add failure and exception handling tests.
   * Effort: Small, Priority: 5.
2. Add input validation tests.
   * Effort: Small, Priority: 4.
3. Verify logger is called for audit.
   * Effort: Small, Priority: 3.
4. Adopt naming convention (Given\_When\_Then) for clarity.
   * Effort: Quick Win, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on EditOrganisationCommand and IOrganisationService.EditOrganisationAsync.
* Contract risk: if org name validation rules change, test must be updated.

**🗄️ Data Model Notes**

* Organisation entity (OrganisationId, Name) must enforce uniqueness and not null constraints.
* DB-level audit logging is recommended for org updates.

**✅ Confidence**

**High** – File fully available; narrow coverage, but easy to extend.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/EditProfileCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~430

**🧾 ELI5**

This file tests the handler that updates a user’s profile. It verifies that profile edits succeed when valid inputs are provided and fail when the user service rejects the update.

**🎯 Purpose and Role**

* Validates **Application layer command handler** EditProfileCommandHandler.
* Ensures correct interaction with IUserService.
* Confirms that both success and failure cases for updating a user profile are handled.
* Protects RoadmApp’s **user profile management workflow** from regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IUserService.
  + ILogger<EditProfileCommandHandler>.
* **Handler under test:** EditProfileCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Succeed\_When\_Update\_Is\_Successful**
    - Creates EditProfileCommand with UserId, FullName = "New Name", Email = "new@example.com".
    - Mocks EditProfileAsync → returns Success(true).
    - Asserts:
      * IsSuccess = true.
      * Error = null.
  + **Handle\_Should\_Fail\_When\_UserService\_Fails**
    - Creates EditProfileCommand with UserId, FullName = "Name", Email = "bademail".
    - Mocks EditProfileAsync → returns Failure("Profile update failed.").
    - Asserts:
      * IsSuccess = false.
      * Error contains "failed".

**⚠️ Error Handling & Validation**

* ✅ Covers success and failure scenarios.
* ❌ Missing tests for:
  + Invalid/malformed email format.
  + Empty FullName.
  + Null/empty UserId.
  + Service throwing exception.

**🔐 Security Review**

* ✅ No secrets or PII leaked in test code.
* ⚠️ Tests rely on raw email strings → no coverage for sanitisation or validation enforcement.
* ❌ No checks for whether profile updates trigger audit logging (security requirement).

**⚡ Performance & Reliability**

* Lightweight (mock-based).
* Async/await correctly implemented.
* No bulk or concurrency scenarios tested (e.g., multiple simultaneous updates).

**📊 Observability**

* Logger is mocked but not asserted.
* No verification of log entries for success/failure.

**🧪 Testability & Coverage**

**Covered:**

* Successful profile update.
* Failed profile update.

**Missing (should add):**

1. **Validation:** Malformed email, empty name → should fail.
2. **Failure path:** Service throws exception.
3. **Edge:** Profile update with Guid.Empty.
4. **Observability:** Verify logs for both paths.
5. **Security:** Verify updates restricted to authenticated users (not covered here).

**🧹 Code Smells**

* **Medium:** Only two tests → limited coverage.
* **Low:** Logger unused.
* **Info:** Test names descriptive but not in Given\_When\_Then form.

**🔧 Refactoring Suggestions**

1. Add validation tests for email/name.
   * Effort: Small, Priority: 5.
2. Add exception-handling test (service throws).
   * Effort: Small, Priority: 4.
3. Verify logger usage.
   * Effort: Small, Priority: 3.
4. Adopt consistent test naming style.
   * Effort: Quick Win, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on EditProfileCommand and IUserService.EditProfileAsync.
* Contract changes (adding/removing fields) will break tests.
* DB schema for Users must enforce email uniqueness.

**🗄️ Data Model Notes**

* Updates map to RoadmappUser entity.
* DB should enforce non-null email + proper constraints.
* Tests assume application layer enforces validation before persistence.

**✅ Confidence**

**High** – File reviewed fully; strong starting point, but more validation and observability tests needed.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/ForgotPasswordCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~400

**🧾 ELI5**

This file tests the handler that manages the “forgot password” flow. It checks that when a valid email exists, a password reset process is initiated, and that even for non-existent emails, the system still responds successfully (to avoid account enumeration).

**🎯 Purpose and Role**

* Validates **Application layer command handler** ForgotPasswordCommandHandler.
* Ensures correct use of IUserService.SendPasswordResetAsync.
* Confirms security principle: always return success regardless of email existence.
* Protects RoadmApp’s **password reset workflow** against regressions and leaks.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IUserService.
  + ILogger<ForgotPasswordCommandHandler>.
* **Handler under test:** ForgotPasswordCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Return\_Success\_And\_Generic\_Message\_When\_Email\_Exists**
    - Mocks SendPasswordResetAsync("user@example.com") → Success(true).
    - Asserts: IsSuccess = true, error is null.
  + **Handle\_Should\_Return\_Success\_And\_Generic\_Message\_When\_Email\_Does\_Not\_Exist**
    - Mocks SendPasswordResetAsync("unknown@example.com") → Success(true).
    - Asserts: IsSuccess = true, error is null.
    - Ensures same behaviour as valid case (prevents leaking which accounts exist).

**⚠️ Error Handling & Validation**

* ✅ Tests confirm enumeration-safe behaviour.
* ❌ Missing scenarios:
  + Service returns failure (e.g., DB error, email server unavailable).
  + Service throws exception.
  + Command with empty/invalid email.

**🔐 Security Review**

* ✅ Tests enforce **enumeration resistance** → always success response.
* ✅ No secrets or PII leaked.
* ⚠️ No coverage for audit logging (important for password reset).
* ❌ No validation for email format.

**⚡ Performance & Reliability**

* Lightweight and fast (mock-based).
* Async/await used correctly.
* No resilience scenarios tested (e.g., retries on email service failure).

**📊 Observability**

* Logger injected but not verified.
* No test coverage of log entries (e.g., warning on reset attempt).

**🧪 Testability & Coverage**

**Covered:**

* Valid email request.
* Non-existent email request.

**Missing (should add):**

1. **Failure path:** Service returns failure → handler should wrap failure.
2. **Exception path:** Service throws exception.
3. **Validation:** Empty or malformed email.
4. **Observability:** Logger records attempt regardless of email existence.

**🧹 Code Smells**

* **Medium:** Only two tests → coverage is limited.
* **Low:** Logger unused.
* **Info:** Test names are verbose but not consistent with BDD naming.

**🔧 Refactoring Suggestions**

1. Add test for service failure (e.g., email provider down).
   * Effort: Small, Priority: 5.
2. Add validation tests for empty/invalid email format.
   * Effort: Small, Priority: 4.
3. Verify logger usage for observability/audit trail.
   * Effort: Small, Priority: 3.
4. Adopt Given\_When\_Then naming convention.
   * Effort: Quick Win, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on ForgotPasswordCommand and IUserService.SendPasswordResetAsync.
* Any change in password reset contract will require test updates.
* Behaviour (always success) is part of **security contract** → must remain stable.

**🗄️ Data Model Notes**

* Indirectly linked to PasswordResetToken entity.
* DB should store hashed reset tokens with expiry.
* These tests don’t verify persistence, only service call.

**✅ Confidence**

**High** – Full file reviewed; confirms enumeration-safe behaviour, but lacks failure, validation, and observability coverage.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/GetDashboardQueryHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Query handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~350

**🧾 ELI5**

This file tests the handler that retrieves dashboard data for a user. It checks that when a valid request is made, the handler returns a populated DashboardViewModel with the expected values.

**🎯 Purpose and Role**

* Validates **Application layer query handler** GetDashboardQueryHandler.
* Ensures correct integration with IDashboardService.
* Confirms that user dashboard data is mapped properly from the service to the view model.
* Protects RoadmApp’s **dashboard feature** against regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IDashboardService.
* **Handler under test:** GetDashboardQueryHandler.
* **Test case:**
  + Creates GetDashboardQuery with UserId.
  + Prepares DashboardViewModel with:
    - CurrentUser = "Test User"
    - TotalTenants = 3
    - RecentApiCalls = 7
    - Notifications = ["Welcome!"]
  + Mocks GetDashboardDataAsync to return the view model.
  + Asserts:
    - IsSuccess = true.
    - Returned value matches expected view model.

**⚠️ Error Handling & Validation**

* ✅ Success path tested.
* ❌ Missing scenarios:
  + Service returns null or empty data.
  + Service throws exception.
  + Invalid UserId (e.g., Guid.Empty).

**🔐 Security Review**

* ✅ No secrets or PII leaked in tests.
* ⚠️ Dashboard may include user-specific sensitive data → test doesn’t check sanitisation or access control enforcement.

**⚡ Performance & Reliability**

* Lightweight test (mock-based).
* Async/await correctly handled.
* No coverage for performance under large dashboard datasets.

**📊 Observability**

* Logger not injected/tested.
* No assertions for log messages or metrics.

**🧪 Testability & Coverage**

**Covered:**

* Valid request returns expected dashboard data.

**Missing (should add):**

1. **Failure path:** Service returns failure or null.
2. **Exception path:** Service throws exception.
3. **Validation:** Empty/invalid UserId.
4. **Edge:** Dashboard with no tenants or API calls.
5. **Observability:** Verify logs on request.

**🧹 Code Smells**

* **Medium:** Only one test case → narrow coverage.
* **Low:** Logger absent.
* **Info:** Test name not in BDD style.

**🔧 Refactoring Suggestions**

1. Add null/failure response tests.
   * Effort: Small, Priority: 5.
2. Add exception-handling test.
   * Effort: Small, Priority: 4.
3. Add edge-case dashboard test (zero tenants, zero calls).
   * Effort: Small, Priority: 3.
4. Adopt BDD-style test names.
   * Effort: Quick Win, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on GetDashboardQuery and IDashboardService.GetDashboardDataAsync.
* Any schema changes in DashboardViewModel will require test updates.

**🗄️ Data Model Notes**

* Dashboard aggregates data from multiple entities (users, tenants, logs).
* DB schema changes (e.g., dropping tenants or logs) may affect dashboard logic indirectly.

**✅ Confidence**

**High** – Full file reviewed; core success path covered, but needs failure, validation, and observability tests.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/GetOrganisationInfoQueryHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Query handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~320

**🧾 ELI5**

This file tests the handler that retrieves organisation details. It verifies that when a valid organisation ID is supplied, the service returns correct information, and the handler outputs a success response.

**🎯 Purpose and Role**

* Validates **Application layer query handler** GetOrganisationInfoQueryHandler.
* Ensures correct integration with IOrganisationService.
* Confirms that organisation info is mapped correctly to the DTO.
* Protects RoadmApp’s **organisation management feature** against regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IOrganisationService.
* **Handler under test:** GetOrganisationInfoQueryHandler.
* **Test case:**
  + Prepares an OrganisationInfoDto with:
    - OrganisationId (Guid).
    - OrganisationName = "Org".
    - UserCount = 2.
    - CreatedAt = DateTime.UtcNow.AddYears(-1).
  + Mocks GetInfoAsync → returns Success(dto).
  + Creates query with OrganisationId.
  + Calls Handle and asserts:
    - IsSuccess = true.
    - Value matches expected DTO.

**⚠️ Error Handling & Validation**

* ✅ Success case covered.
* ❌ Missing:
  + Service returns failure (organisation not found).
  + Service throws exception.
  + Invalid OrganisationId (e.g., empty GUID).

**🔐 Security Review**

* ✅ No secrets or PII in test.
* ⚠️ Organisation info may contain sensitive business data in production; no audit logging assertions here.

**⚡ Performance & Reliability**

* Lightweight mock test.
* Async/await handled correctly.
* No concurrency/scale cases tested (e.g., large number of users).

**📊 Observability**

* No logger injected or verified.
* No auditability checks for sensitive entity access.

**🧪 Testability & Coverage**

**Covered:**

* Valid organisation ID returns expected info.

**Missing (should add):**

1. **Failure path:** Service returns not found.
2. **Exception path:** Service throws exception.
3. **Validation:** Empty GUID should fail.
4. **Observability:** Verify logging of request.

**🧹 Code Smells**

* **Medium:** Only one test case → narrow coverage.
* **Low:** No logging tested.
* **Info:** Test name could be more descriptive (Should\_ReturnOrganisationInfo\_When\_Found).

**🔧 Refactoring Suggestions**

1. Add not found/failure path test.
   * Effort: Small, Priority: 5.
2. Add exception handling test.
   * Effort: Small, Priority: 4.
3. Add input validation test (Guid.Empty).
   * Effort: Small, Priority: 3.
4. Improve test naming to BDD style.
   * Effort: Quick Win, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on GetOrganisationInfoQuery and IOrganisationService.GetInfoAsync.
* DTO contract (OrganisationInfoDto) must remain stable or tests will break.

**🗄️ Data Model Notes**

* DTO reflects Domain entity Organisation.
* Schema should enforce non-null OrganisationName.
* UserCount should be consistent with actual users linked in DB.

**✅ Confidence**

**High** – Full file available, confirms handler works, but coverage is too limited.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/RegisterUserCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~700

**🧾 ELI5**

This file tests the handler responsible for registering new users. It verifies that new users can be created successfully, and that the handler fails when the username or email already exists.

**🎯 Purpose and Role**

* Validates **Application layer command handler** RegisterUserCommandHandler.
* Ensures correct integration with IUserService.RegisterAsync.
* Confirms that both success and failure scenarios for user registration are handled.
* Protects RoadmApp’s **user onboarding workflow** from regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IUserService.
  + ILogger<RegisterUserCommandHandler>.
* **Handler under test:** RegisterUserCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Return\_Success\_When\_User\_Is\_Created**
    - Creates RegisterUserCommand with Username, Email, Password, FullName.
    - Prepares a RoadmappUser with IsActive = true.
    - Mocks RegisterAsync → returns Success(user).
    - Asserts:
      * Username, Email, FullName match.
      * Message = "User created successfully."
      * RequiresPasswordSetup = false.
      * RequiresEmailVerification = false.
  + **Handle\_Should\_Return\_Error\_When\_Registration\_Fails**
    - Creates RegisterUserCommand with duplicate details.
    - Mocks RegisterAsync → returns Failure("Username or email already exists.").
    - Asserts:
      * Message contains failure text.
      * RequiresPasswordSetup = false.
      * RequiresEmailVerification = false.

**⚠️ Error Handling & Validation**

* ✅ Success and duplicate failure cases tested.
* ❌ Missing:
  + Empty/null username, email, password.
  + Invalid email format.
  + Weak password rejected.
  + Service throws exception.

**🔐 Security Review**

* ✅ No secrets or PII exposed in tests.
* ✅ Confirms that duplicate usernames/emails are rejected.
* ❌ No coverage for password security enforcement (length, complexity, hashing).
* ❌ No tests verifying that new users default to least-privilege role.

**⚡ Performance & Reliability**

* Lightweight, mock-based tests.
* Async/await handled correctly.
* No coverage for bulk registrations or race conditions (e.g., two users registering simultaneously with the same email).

**📊 Observability**

* Logger injected but not asserted.
* No verification that logs capture registration attempts or failures.

**🧪 Testability & Coverage**

**Covered:**

* Successful registration.
* Failure on duplicate user/email.

**Missing (should add):**

1. **Validation:** Empty or malformed inputs.
2. **Security:** Weak password enforcement.
3. **Exception path:** Service throws exception.
4. **Concurrency:** Duplicate registration under load.
5. **Observability:** Verify logger entries.

**🧹 Code Smells**

* **Medium:** Limited coverage, only two scenarios.
* **Low:** Logger unused.
* **Info:** Test names are descriptive but inconsistent in length/detail.

**🔧 Refactoring Suggestions**

1. Add validation tests for inputs.
   * Effort: Small, Priority: 5.
2. Add password complexity tests.
   * Effort: Small, Priority: 4.
3. Add exception-handling test.
   * Effort: Small, Priority: 4.
4. Verify logger usage.
   * Effort: Small, Priority: 3.
5. Add concurrency test for duplicate registration.
   * Effort: Medium, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on RegisterUserCommand and IUserService.RegisterAsync.
* DTO contract (UserRegistrationRequest) and entity (RoadmappUser) must remain stable.

**🗄️ Data Model Notes**

* New users map to Users table.
* DB schema must enforce uniqueness on Email and Username.
* Password stored as hash (not tested here).

**✅ Confidence**

**High** – File fully available; covers main cases, but needs stronger validation, security, and observability tests.

**File Review**

**File:** Tests/Application/Features/Users/Commands/Handlers/ResetPasswordCommandHandlerTests.cs  
**Layer/Type:** Tests – Application (Feature/Command handler test)  
**Status:** Reviewed  
**Tokens (approx.):** ~550

**🧾 ELI5**

This file tests the handler responsible for resetting user passwords using password reset tokens. It checks that valid tokens allow a password reset, while expired tokens cause failure.

**🎯 Purpose and Role**

* Validates **Application layer command handler** ResetPasswordCommandHandler.
* Ensures correct integration with:
  + IPasswordResetTokenRepository → validates tokens.
  + IUserService → updates user passwords.
* Confirms that both success and failure scenarios for password resets are handled.
* Protects RoadmApp’s **password recovery workflow** from regressions.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IPasswordResetTokenRepository.
  + IUserService.
  + ILogger<ResetPasswordCommandHandler>.
* **Handler under test:** ResetPasswordCommandHandler.
* **Test cases:**
  + **Handle\_Should\_Succeed\_For\_Valid\_Token\_And\_New\_Password**
    - Prepares valid PasswordResetToken.
    - Mocks repository FindAsync → returns token.
    - Mocks ChangePasswordAsync → returns success.
    - Asserts: IsSuccess = true, no error.
  + **Handle\_Should\_Return\_Failure\_For\_Expired\_Token**
    - Prepares expired token (ExpiresAt < DateTime.UtcNow).
    - Mocks repository → returns token.
    - Executes handler → result fails.
    - Asserts: IsSuccess = false, error contains "expired".

**⚠️ Error Handling & Validation**

* ✅ Covers success and expired token failure.
* ❌ Missing:
  + Token not found.
  + Token already used.
  + Service throws exception.
  + Empty/weak new password.

**🔐 Security Review**

* ✅ Tests confirm tokens must be valid and not expired.
* ✅ Uses hashed tokens (BCrypt.Net.BCrypt.HashPassword("raw-token")).
* ❌ No tests for brute force prevention (rate-limiting reset attempts).
* ❌ No coverage for password strength enforcement.

**⚡ Performance & Reliability**

* Lightweight test (mock-based).
* Async/await used properly.
* No concurrency tests (multiple resets in parallel).

**📊 Observability**

* Logger injected but not verified.
* No assertions for logs or audit events (critical for password resets).

**🧪 Testability & Coverage**

**Covered:**

* Valid reset with good token.
* Expired token rejection.

**Missing (should add):**

1. **Not found:** Token does not exist.
2. **Used:** Token already marked as used.
3. **Failure path:** ChangePasswordAsync fails.
4. **Validation:** Weak/empty new password.
5. **Observability:** Verify logging of attempts.

**🧹 Code Smells**

* **Medium:** Only two test cases → limited coverage.
* **Low:** Logger unused.
* **Info:** Test names descriptive but inconsistent (Handle\_Should\_\*).

**🔧 Refactoring Suggestions**

1. Add test for missing/used tokens.
   * Effort: Small, Priority: 5.
2. Add test for weak/empty passwords.
   * Effort: Small, Priority: 4.
3. Add test for ChangePasswordAsync failure.
   * Effort: Small, Priority: 4.
4. Verify logging/audit calls.
   * Effort: Small, Priority: 3.
5. Add concurrency/resilience scenario (parallel resets).
   * Effort: Medium, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on ResetPasswordCommand, IPasswordResetTokenRepository, and IUserService.ChangePasswordAsync.
* Token contract (hashing, expiry) must remain stable.

**🗄️ Data Model Notes**

* Maps to PasswordResetToken entity in Domain.
* DB must enforce unique tokens, expiry checks, and used/unused state.
* Tests assume Infrastructure layer handles token hashing.

**✅ Confidence**

**High** – File fully reviewed; validates core reset scenarios but lacks full coverage of failure and edge cases.

**File Review**

**File:** Tests/Common/ResultTests.cs  
**Layer/Type:** Tests – Common (Utility test)  
**Status:** Reviewed  
**Tokens (approx.):** ~30 (very small file)

**🧾 ELI5**

This file is supposed to test the Result utility type (used widely across the Application layer to wrap success/failure outcomes), but it currently contains **only metadata comments** and no actual test cases.

**🎯 Purpose and Role**

* Intended to validate the Result<T> utility class in the **Common** library.
* Should confirm that success and failure outcomes behave correctly (e.g., IsSuccess, Error, Value).
* Currently **empty** — no tests implemented.

**🔍 Detailed Breakdown**

* File only contains SPDX license header and metadata (created: 2025-06-16).
* No namespaces, classes, or test methods defined.

**⚠️ Error Handling & Validation**

* None present — tests are missing entirely.

**🔐 Security Review**

* ✅ No secrets or PII.
* ❌ Lack of tests means potential for silent failures in Result<T> handling, which could have security implications if errors are not propagated correctly.

**⚡ Performance & Reliability**

* N/A — no tests.
* A proper suite should validate performance-neutral behaviour of Result<T>.

**📊 Observability**

* N/A — no logging or assertions.

**🧪 Testability & Coverage**

**Current coverage:** 0% for Result<T> utility.

**Recommended test cases:**

1. **Success case:** Result.Success("value") → IsSuccess = true, Value = "value", Error = null.
2. **Failure case:** Result.Failure("error") → IsSuccess = false, Error = "error", Value = default.
3. **Generic typing:** Result<int>.Success(42) should work correctly.
4. **Chaining:** Ensure results can be passed across layers.
5. **Edge case:** Null value in success → verify behaviour.

**🧹 Code Smells**

* **Critical:** Placeholder only — no actual tests.

**🔧 Refactoring Suggestions**

1. Implement a full test suite for Result<T> covering success/failure, generics, and edge cases.
   * Effort: Small, Priority: 5.
2. Add negative tests to confirm correct error propagation.
   * Effort: Small, Priority: 4.
3. Consider adding parameterised tests for multiple data types.
   * Effort: Small, Priority: 3.

**📜 Contracts & Compatibility**

* Result<T> is part of the **Application contract**.
* Must remain stable, as many handlers return Result<T>.
* Any changes to this type would break all MediatR handlers.

**🗄️ Data Model Notes**

* Not applicable (utility class, no schema).

**✅ Confidence**

**High** – Full file reviewed; confirmed it is empty and needs implementation.

**File Review**

**File:** Tests/Common/TestWebApplicationFactory.cs  
**Layer/Type:** Tests – Common (Integration Test Support)  
**Status:** Reviewed  
**Tokens (approx.):** ~230

**🧾 ELI5**

This file sets up a test version of the RoadmApp web application. It uses the standard ASP.NET test host but forces it into a "Testing" environment and loads test-specific configuration files.

**🎯 Purpose and Role**

* Provides a **shared WebApplicationFactory** for integration and end-to-end tests.
* Ensures tests run with:
  + The "Testing" environment.
  + appsettings.json (base settings).
  + appsettings.Testing.json (test overrides).
* Allows test projects to spin up a lightweight in-memory web server for API tests.

**🔍 Detailed Breakdown**

* Inherits from WebApplicationFactory<Program>.
* Overrides ConfigureWebHost:
  + Calls UseEnvironment("Testing").
  + Configures IConfigurationBuilder:
    - Adds appsettings.json (optional, reloadOnChange: false).
    - Adds appsettings.Testing.json (optional, reloadOnChange: false).
    - Adds an **in-memory dictionary** (currently empty → placeholder for test overrides).
* Makes use of Microsoft.AspNetCore.Mvc.Testing infrastructure.

**⚠️ Error Handling & Validation**

* No explicit error handling here (expected — framework-managed).
* If appsettings.Testing.json is missing, tests may fail silently without overrides.

**🔐 Security Review**

* ✅ No secrets hard-coded.
* ⚠️ Tests depend on config files → must ensure these do not include production credentials.
* Best practice: keep appsettings.Testing.json limited to safe values (in-memory DBs, fake keys).

**⚡ Performance & Reliability**

* Starts full ASP.NET host for tests (slower than unit tests, but necessary for integration).
* Reload disabled (reloadOnChange = false) → improves test consistency.
* No dependency injection overrides → may need extension for service mocking.

**📊 Observability**

* No logging adjustments for tests.
* Test environment logs may default to normal levels (risk of excessive output).

**🧪 Testability & Coverage**

* Provides foundation for integration tests but itself is not tested.
* Should be combined with:
  + CustomWebApplicationFactory (for service overrides).
  + Auth helpers (mock authentication).
* Suggested additional coverage:
  + Verify that "Testing" environment is applied.
  + Confirm fallback when config files missing.
  + Allow injection of test services (e.g., mock Redis, in-memory DB).

**🧹 Code Smells**

* **Info:** Currently only loads settings — no DI overrides for test-specific services.
* **Low:** Empty in-memory dictionary → placeholder but unused.

**🔧 Refactoring Suggestions**

1. Add support for **service overrides** (e.g., mock email, mock Redis).
   * Effort: Medium, Priority: 5.
2. Add fallback defaults if appsettings.Testing.json not present.
   * Effort: Small, Priority: 4.
3. Adjust logging levels for cleaner test output.
   * Effort: Small, Priority: 3.

**📜 Contracts & Compatibility**

* Acts as **contract** between test framework and Web host.
* Must remain compatible with Program.cs entry point.
* Breaking changes in Program (DI setup, host config) may require updates.

**🗄️ Data Model Notes**

* N/A (no schema — config loader only).

**✅ Confidence**

**High** – File fully available; solid integration test foundation but lacks service override flexibility.

**File Review**

**File:** Tests/Domain/RoadmappUserTests.cs  
**Layer/Type:** Tests – Domain (Entity test)  
**Status:** Reviewed  
**Tokens (approx.):** ~25 (very small file)

**🧾 ELI5**

This file is supposed to contain unit tests for the RoadmappUser domain entity, but it currently has **only comments and metadata** and no test logic.

**🎯 Purpose and Role**

* Intended to validate the **RoadmappUser entity** in the Domain layer.
* Should test business rules and properties such as:
  + Required fields (UserId, Username, Email, PasswordHash).
  + Active/inactive status logic.
  + Integration with roles/organisations.
* Currently **empty** — no actual tests implemented.

**🔍 Detailed Breakdown**

* Only contains SPDX license header and metadata (created: 2025-06-16).
* No test methods or namespaces defined.

**⚠️ Error Handling & Validation**

* None present — entity is untested.

**🔐 Security Review**

* ✅ No secrets or PII in file.
* ❌ Lack of tests means risk of silent failures in user entity validation (critical for authentication).

**⚡ Performance & Reliability**

* N/A (no tests).

**📊 Observability**

* N/A (no logging or assertions).

**🧪 Testability & Coverage**

**Current coverage:** 0% for RoadmappUser.

**Recommended test cases:**

1. **Happy path:** Create user with valid Username, Email, PasswordHash → properties set correctly.
2. **Negative:** Empty Username or Email → should be rejected.
3. **Negative:** Empty PasswordHash → must fail.
4. **Edge:** Set IsActive = false → ensure disabled users cannot be authenticated.
5. **Integration:** Add roles/org associations → confirm correct linking.

**🧹 Code Smells**

* **Critical:** Placeholder file with no tests.

**🔧 Refactoring Suggestions**

1. Implement tests for RoadmappUser covering required properties and constraints.
   * Effort: Small, Priority: 5.
2. Add tests for IsActive status.
   * Effort: Small, Priority: 4.
3. Add role/organisation linking tests (via UserOrgRole).
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* RoadmappUser is a **core domain entity**.
* Any changes to property definitions affect authentication, authorisation, and persistence.
* Tests here would act as contract validation for the Domain layer.

**🗄️ Data Model Notes**

* Maps to Users table.
* Schema must enforce:
  + Unique Username and Email.
  + Non-null PasswordHash.
  + Boolean IsActive.

**✅ Confidence**

**High** – Full file reviewed; confirmed as a placeholder with missing tests for a critical domain entity.

**File Review**

**File:** Tests/Helpers/CustomWebApplicationFactory.cs  
**Layer/Type:** Tests – Helpers (Integration Test Support)  
**Status:** Reviewed  
**Tokens (approx.):** ~480

**🧾 ELI5**

This file creates a custom version of the RoadmApp test web host. It replaces the normal authentication with a fake “Test” scheme so that integration tests can run without real login, while still loading the "Testing" configuration.

**🎯 Purpose and Role**

* Extends ASP.NET’s WebApplicationFactory<Program>.
* Provides a **controlled environment** for integration and end-to-end tests.
* Ensures:
  + "Testing" environment is used.
  + Test configuration file (appsettings.Testing.json) is loaded.
  + Real authentication is replaced with a test handler (TestAuthHandler).

**🔍 Detailed Breakdown**

* **Overrides ConfigureWebHost:**
  + Forces environment to "Testing".
  + Loads test-specific config from appsettings.Testing.json.
  + Ensures relative path resolution works when running in test host.
* **Service overrides:**
  + Removes any existing IAuthenticationService registrations.
  + Adds custom authentication with default scheme "Test".
  + Configures AuthenticationSchemeOptions to use TestAuthHandler.
* **Overrides CreateHost:**
  + Ensures content root is set to project root (".").
  + Calls base host creation.

**⚠️ Error Handling & Validation**

* ❌ No explicit error handling → if appsettings.Testing.json is missing, test host may fail silently.
* ❌ No validation that test authentication is actually applied.

**🔐 Security Review**

* ✅ Replaces real authentication with a fake handler → safe for test environment.
* ⚠️ Risk: if accidentally used in production builds, would bypass security entirely.
* Recommendation: enforce compile-time separation (#if DEBUG or explicit test project scope).

**⚡ Performance & Reliability**

* Suitable for integration tests, but full web host startup is slower than unit tests.
* Test auth avoids external calls, so tests remain reliable.

**📊 Observability**

* No logger overrides or test log sinks provided.
* Integration tests will log normally unless further configured.

**🧪 Testability & Coverage**

* Makes integration tests **easier** by skipping real auth.
* Should be tested indirectly by running API-level tests with/without authentication.

**Suggested missing cases for broader coverage:**

1. Validate that TestAuthHandler is active during tests.
2. Confirm fallback if config file missing.
3. Test overrides for other services (e.g., Redis, SMTP) if required.

**🧹 Code Smells**

* **Medium:** Relies on relative paths for appsettings.Testing.json. Fragile on CI/CD environments.
* **Low:** Logger not overridden for cleaner test output.
* **Info:** All services except auth are real → may cause unwanted external dependencies in tests.

**🔧 Refactoring Suggestions**

1. Add **guard clause** to throw if appsettings.Testing.json not found.
   * Effort: Small, Priority: 5.
2. Add **test DI overrides** (e.g., in-memory DB, fake Redis).
   * Effort: Medium, Priority: 4.
3. Configure test logging sink (less noise, structured logs).
   * Effort: Small, Priority: 3.
4. Enforce build-scope separation (#if TESTING).
   * Effort: Small, Priority: 5.

**📜 Contracts & Compatibility**

* Contract between test harness and the ASP.NET host.
* Must remain compatible with Program.cs.
* If auth defaults change in production code, this test factory must be updated.

**🗄️ Data Model Notes**

* N/A (test infra only).

**✅ Confidence**

**High** – File fully available; strong test infrastructure, but should improve DI overrides and config validation.

**File Review**

**File:** Tests/Helpers/IntegrationTestAuthHelper.cs  
**Layer/Type:** Tests – Helpers (Authentication Stub for Integration Tests)  
**Status:** Reviewed  
**Tokens (approx.):** ~380

**🧾 ELI5**

This file defines a fake authentication handler for integration tests. Instead of requiring real OAuth or identity providers, it automatically signs in a test user with a configurable role.

**🎯 Purpose and Role**

* Provides a **custom AuthenticationHandler** for test scenarios.
* Used in conjunction with CustomWebApplicationFactory.
* Ensures integration tests can simulate authenticated requests without real login flows.
* Supports role-based test scenarios by injecting a role via TestAuthRoleProvider.

**🔍 Detailed Breakdown**

* **Class:** TestAuthHandler inherits from AuthenticationHandler<AuthenticationSchemeOptions>.
* **Constructor:** Accepts options, logger, encoder.
* **Core method:** HandleAuthenticateAsync().
  + Fetches role from TestAuthRoleProvider (registered in test DI).
  + If role is missing/null → returns NoResult() (causes a 401 Unauthorized).
  + Builds a set of Claims:
    - Name = "TestUser".
    - Role = <injected role>.
    - NameIdentifier = Guid.NewGuid().ToString().
  + Wraps them in ClaimsIdentity and ClaimsPrincipal.
  + Issues an AuthenticationTicket with scheme "Test".
  + Returns AuthenticateResult.Success(ticket).

**⚠️ Error Handling & Validation**

* ✅ Returns NoResult() if role not provided (good fail-safe).
* ❌ No validation of claim inputs.
* ❌ No logging of failures/success.

**🔐 Security Review**

* ✅ Only used in test scope (safe).
* ✅ Role injection makes it flexible for role-based access tests.
* ⚠️ If accidentally left enabled in production, would bypass authentication completely.
* Recommendation: enforce build-scope separation.

**⚡ Performance & Reliability**

* Lightweight, runs in-memory.
* Always generates a new Guid for NameIdentifier → avoids collisions across tests.
* Reliable for role-based test coverage.

**📊 Observability**

* Logger is injected but unused.
* No audit trail for when authentication succeeds/fails in tests.

**🧪 Testability & Coverage**

**Provides the foundation** for:

* Testing authenticated endpoints without OAuth complexity.
* Simulating different user roles.
* Handling anonymous scenarios (role not set).

**Missing (should add):**

1. Verify behaviour when Role = null → should trigger 401.
2. Verify claims propagation in an integration test (e.g., controller sees User.Identity.Name).
3. Verify multiple roles (if supported).

**🧹 Code Smells**

* **Low:** Logger unused.
* **Info:** Always assigns "TestUser" as name — may limit test variety.

**🔧 Refactoring Suggestions**

1. Use logger to record authentication attempts.
   * Effort: Small, Priority: 3.
2. Allow configurable Name claim for more flexible test coverage.
   * Effort: Small, Priority: 3.
3. Enforce compile-time/test-only usage.
   * Effort: Small, Priority: 5.

**📜 Contracts & Compatibility**

* Contract with ASP.NET Core authentication system.
* Must remain compatible with "Test" authentication scheme configured in CustomWebApplicationFactory.

**🗄️ Data Model Notes**

* Not applicable (no DB persistence).

**✅ Confidence**

**High** – Full file reviewed; strong test support for authentication, but needs minor improvements for observability and configurability.

**File Review**

**File:** Tests/Helpers/TestAuthRoleProvider.cs  
**Layer/Type:** Tests – Helpers (Authentication Role Provider)  
**Status:** Reviewed  
**Tokens (approx.):** ~80

**🧾 ELI5**

This file defines a simple class that stores a role string for test authentication. It allows integration tests to simulate different user roles by setting a property before making authenticated requests.

**🎯 Purpose and Role**

* Provides a **shared role provider** for TestAuthHandler.
* Used in integration tests to control the role claim injected into the authentication pipeline.
* Enables test cases like:
  + "Admin" role access.
  + "User" role restrictions.

**🔍 Detailed Breakdown**

* **Class:** TestAuthRoleProvider.
* **Property:**
  + string? Role { get; set; } → nullable string representing current test role.
* **Usage:**
  + Before test execution, set Role = "Admin" (or other).
  + TestAuthHandler will read this value when authenticating requests.

**⚠️ Error Handling & Validation**

* ❌ No validation (any string can be set, even empty or invalid).
* ❌ No default role assigned.

**🔐 Security Review**

* ✅ Safe for test scope.
* ⚠️ If misused (e.g., left null), authentication will fail and return 401 Unauthorized.
* ❌ No role normalisation (case-sensitivity issues possible).

**⚡ Performance & Reliability**

* Very lightweight, in-memory property.
* Thread-safety not guaranteed → if multiple parallel tests set different roles, race conditions could occur.

**📊 Observability**

* No logging.

**🧪 Testability & Coverage**

* Provides flexibility for integration tests.
* Should be tested indirectly by asserting that authenticated requests reflect the assigned role.

**Suggested additional tests:**

1. Role not set → request should fail authentication.
2. Role set → authenticated request includes correct claim.
3. Case sensitivity handling ("admin" vs "Admin").

**🧹 Code Smells**

* **Low:** Role stored as plain string → fragile compared to enum.
* **Info:** Thread-unsafe → parallel tests could interfere.

**🔧 Refactoring Suggestions**

1. Change Role to an **enum** (e.g., Admin, User, ReadOnly).
   * Effort: Small, Priority: 4.
2. Add default fallback role (e.g., User).
   * Effort: Small, Priority: 3.
3. Make property thread-safe (e.g., AsyncLocal<string?>).
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* Contract with TestAuthHandler.
* Must remain stable; any renaming or type change breaks integration with the test authentication pipeline.

**🗄️ Data Model Notes**

* N/A (test helper only).

**✅ Confidence**

**High** – File fully reviewed; extremely simple, but should be strengthened for robustness in parallel tests.

**File Review**

**File:** Tests/Infrastructure/External/Xero/XeroEtlServiceTests.cs  
**Layer/Type:** Tests – Infrastructure (External service integration test)  
**Status:** Reviewed  
**Tokens (approx.):** ~600

**🧾 ELI5**

This file tests the ETL (Extract-Transform-Load) service that pulls data from Xero into RoadmApp. It simulates retrieving raw data, staging it, and processing it into operational tables. The tests use mocks for repositories and Python ETL bridges.

**🎯 Purpose and Role**

* Validates the **Xero ETL pipeline service** (XeroEtlService).
* Ensures correct orchestration of:
  + **Raw repository** (IRawXeroPayloadRepository).
  + **Python bridge** (IPythonEtlBridge) for transformation.
  + **Call log service**.
* Confirms the service can run successfully and merge results from multiple stages.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IRawXeroPayloadRepository → persists raw payloads.
  + IPythonEtlBridge → runs ETL Python scripts (RawToLanding, LandingToOds).
  + ICallLogService.
* **Service under test:** XeroEtlService.
* **Test setup:**
  + Configures in-memory IConfiguration with fake Postgres connection string.
  + Prepares ETL profiles: RawToLanding, LandingToOds.
  + Sets up mocks:
    - RawRepo.MergeStagingAsync → returns success.
    - PythonBridge.RunAsync → returns JSON string.
    - CallLogService stubbed.
  + Creates XeroEtlService with mocks.
* **Test case:** RunEtlAsync\_CallsMergeAndAllPythonProfiles
  + Prepares tenant/user IDs.
  + Configures raw repo and bridge mocks to return success.
  + Calls sut.RunEtlAsync(endpoint, tenantId, userId).
  + Asserts that raw repo and bridge mocks were called (verifies orchestration).

**⚠️ Error Handling & Validation**

* ✅ Happy path (end-to-end ETL success) tested.
* ❌ Missing scenarios:
  + Raw repo throws exception.
  + Python bridge fails (script error, bad JSON).
  + Call log service unavailable.
  + Configuration missing or invalid.

**🔐 Security Review**

* ✅ Uses in-memory config for tests (no secrets).
* ⚠️ In production, connection strings must come from Key Vault — tests do not enforce this.
* ❌ No coverage of data sanitisation — raw JSON payloads may contain sensitive data.

**⚡ Performance & Reliability**

* Mock-based → fast tests.
* No stress/load tests (large payloads, many tenants).
* No retry/failure resilience covered.

**📊 Observability**

* Logger injected via NullLogger.
* No assertions for logging or ETL audit events.

**🧪 Testability & Coverage**

**Covered:**

* End-to-end ETL happy path with raw → landing → ODS.

**Missing (should add):**

1. **Failure path:** Python bridge returns error.
2. **Failure path:** Raw repo merge fails.
3. **Validation:** Null/invalid tenant ID.
4. **Edge:** Empty ETL results.
5. **Observability:** Verify call log entries written.

**🧹 Code Smells**

* **Medium:** Only one test → limited coverage.
* **Low:** Hard-coded ETL profile names.
* **Info:** No negative tests for ETL orchestration.

**🔧 Refactoring Suggestions**

1. Add **negative tests** for Python bridge and repo failures.
   * Effort: Small, Priority: 5.
2. Add **edge case tests** (empty data, invalid tenant).
   * Effort: Small, Priority: 4.
3. Verify **logging/call log entries** are written.
   * Effort: Small, Priority: 3.
4. Make ETL profile list configurable (not hardcoded in test).
   * Effort: Medium, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on contracts:
  + IRawXeroPayloadRepository.MergeStagingAsync.
  + IPythonEtlBridge.RunAsync.
* Any signature changes break tests.
* JSON return schema from Python must remain stable.

**🗄️ Data Model Notes**

* ETL stages correspond to DB schemas: raw, landing, ods.
* Repo test confirms merge from staging into persistent store.
* Payload repository must support multi-tenant isolation.

**✅ Confidence**

**High** – File fully reviewed; validates happy path, but missing resilience and negative scenarios.

**File Review**

**File:** Tests/Infrastructure/External/Xero/XeroRawDataSynchronizerTests.cs  
**Layer/Type:** Tests – Infrastructure (External service synchronizer test)  
**Status:** Reviewed  
**Tokens (approx.):** ~750

**🧾 ELI5**

This file tests the Xero raw data synchronizer, which pulls raw JSON from Xero endpoints and persists it in RoadmApp’s database. The test ensures that only active endpoints are processed and that the order of execution is respected.

**🎯 Purpose and Role**

* Validates **XeroRawDataSynchronizer**, which orchestrates pulling raw data.
* Ensures correct integration with:
  + IEndpointConfigService → provides which endpoints to sync.
  + IRawXeroPayloadRepository → stores fetched payloads.
  + IHttpClientFactory → makes calls to Xero.
* Confirms synchronizer filters **active endpoints** and executes them in ETL order.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IEndpointConfigService → returns endpoint configs.
  + IRawXeroPayloadRepository → saves payloads.
  + IHttpClientFactory → fake HTTP client.
  + ICurrentUserContext (mocked to satisfy constructor).
* **Handler under test:** XeroRawDataSynchronizer.
* **Test setup:**
  + Defines endpoints:
    - A (order 1, active).
    - B (order 2, active).
    - C (order 3, active).
    - D\_Inactive (order 4, inactive → should be skipped).
  + Config service returns these endpoints.
  + Repo mock for GetLastFetchedAsync returns null (fresh run).
  + Fake HTTP client always returns 200 OK with empty JSON ({"Items":[]}).
  + Synchronizer instantiated with mocks + test config.
* **Test case:** RunOnceAsync\_RunsOnlyActiveEndpointsInSortOrder
  + Calls RunOnceAsync(tenantId, userId, CancellationToken.None).
  + Asserts:
    - Only active endpoints run.
    - Execution order is A, B, C.
    - Inactive endpoint skipped.
* **Helper class:** FakeHttpMessageHandler → always returns 200 OK with empty JSON.

**⚠️ Error Handling & Validation**

* ✅ Tests inactive endpoint skipping.
* ✅ Tests execution ordering.
* ❌ Missing:
  + Handling of HTTP failures (400/500 responses).
  + Handling of invalid JSON responses.
  + Handling of exceptions in repo writes.
  + Handling of cancelled token.

**🔐 Security Review**

* ✅ No secrets in test.
* ✅ Test avoids real Xero API calls.
* ⚠️ Real synchronizer must handle OAuth tokens and sensitive payloads → not covered here.
* ❌ No test to ensure PII in raw payload is protected/logged correctly.

**⚡ Performance & Reliability**

* Lightweight, mock-driven test.
* No coverage for large batch pulls or rate limiting (important for production).
* No retries tested.

**📊 Observability**

* Logger is injected but not asserted.
* No audit logs tested for endpoint sync attempts.

**🧪 Testability & Coverage**

**Covered:**

* Active endpoint filtering.
* Execution order validation.

**Missing (should add):**

1. **Failure path:** HTTP 500 → synchronizer should log and continue.
2. **Validation:** Inactive endpoint ignored.
3. **Edge:** Endpoint returns empty JSON vs populated JSON.
4. **Exception:** Repo throws error → synchronizer should handle.
5. **Observability:** Logger assertions for sync start/end.
6. **Security:** Test handling of OAuth token expiry.

**🧹 Code Smells**

* **Medium:** Only one scenario tested.
* **Low:** Fake HTTP client always succeeds → no failure path tested.
* **Info:** Endpoints hardcoded inside test → less flexible.

**🔧 Refactoring Suggestions**

1. Add failure-path tests for HTTP errors.
   * Effort: Small, Priority: 5.
2. Add repo failure/exception tests.
   * Effort: Small, Priority: 4.
3. Add tests for retry/backoff handling.
   * Effort: Medium, Priority: 3.
4. Verify logging for observability.
   * Effort: Small, Priority: 2.
5. Parameterise endpoints list for more flexibility.
   * Effort: Small, Priority: 2.

**📜 Contracts & Compatibility**

* Relies on contracts:
  + IEndpointConfigService.GetAllAsync.
  + IRawXeroPayloadRepository.GetLastFetchedAsync.
  + IHttpClientFactory.CreateClient.
* If contracts change (e.g., endpoint DTOs), test must be updated.

**🗄️ Data Model Notes**

* Endpoints map to ETL stages (raw ingestion).
* DB schema should enforce **endpoint uniqueness** by Name.
* Repo should record last fetched timestamp for incremental sync.

**✅ Confidence**

**High** – Full file reviewed; good coverage of ordering and filtering, but lacks resilience, security, and failure testing.

**File Review**

**File:** Tests/Infrastructure/Persistence/DapperTokenRepositoryInterfaceTests.cs  
**Layer/Type:** Tests – Infrastructure (Repository contract test)  
**Status:** Reviewed  
**Tokens (approx.):** ~250

**🧾 ELI5**

This file checks that the DapperTokenRepository properly implements the expected repository interface contract. It doesn’t test the database behaviour, but rather verifies method signatures and interface compliance.

**🎯 Purpose and Role**

* Ensures DapperTokenRepository implements the ITokenRepository.StoreTokenAsync contract.
* Uses reflection to check interface compliance.
* Provides a safeguard against regression if the method is removed or renamed.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), FluentAssertions, Reflection APIs.
* **Test case:** StoreTokenAsync\_exists\_and\_is\_marked\_obsolete
  + Uses reflection to fetch StoreTokenAsync method on DapperTokenRepository.
  + Asserts that it is not null (i.e., method exists).
  + Also checks for [Obsolete] attribute.
  + Confirms this method is deprecated in favour of StoreTokensAsync.

**⚠️ Error Handling & Validation**

* No runtime error handling tested (e.g., DB failures).
* Purely structural test (existence + attribute).

**🔐 Security Review**

* ✅ No secrets or PII.
* ❌ Does not validate security of token storage (encryption, hashing). That responsibility lies in Infrastructure and DB tests.

**⚡ Performance & Reliability**

* Lightweight, runs reflection only.
* Reliable test for contract compliance.

**📊 Observability**

* N/A — no logging in this test.

**🧪 Testability & Coverage**

**Covered:**

* Existence of StoreTokenAsync method.
* Confirmation that it is marked [Obsolete].

**Missing (should add):**

1. Verify new replacement method StoreTokensAsync exists.
2. Integration tests for actual DB persistence.
3. Negative test for calling obsolete method (warning expected).

**🧹 Code Smells**

* **Low:** Reflection-based tests can be brittle.
* **Info:** Only checks structure, not behaviour.

**🔧 Refactoring Suggestions**

1. Add test to confirm StoreTokensAsync exists and is not obsolete.
   * Effort: Small, Priority: 5.
2. Add integration tests that persist tokens via repository.
   * Effort: Medium, Priority: 4.
3. Add contract enforcement test for async return type (must return Task).
   * Effort: Small, Priority: 3.

**📜 Contracts & Compatibility**

* Confirms DapperTokenRepository adheres to ITokenRepository.
* Contract risk: if StoreTokenAsync is removed without replacement, tests fail.
* Maintains backward compatibility check during migration.

**🗄️ Data Model Notes**

* Tokens map to TokenRecord entity.
* DB schema must persist EncryptedAccessToken and EncryptedRefreshToken.
* This test does not validate schema-level integrity.

**✅ Confidence**

**High** – File fully reviewed; provides structural contract assurance, but no behavioural coverage.

**File Review**

**File:** Tests/Infrastructure/Persistence/DapperUserRepositoryTests.cs  
**Layer/Type:** Tests – Infrastructure (Repository test placeholder)  
**Status:** Reviewed  
**Tokens (approx.):** ~25 (very small file)

**🧾 ELI5**

This file is supposed to contain tests for the DapperUserRepository, but it only has comments and metadata — no actual test cases are implemented.

**🎯 Purpose and Role**

* Intended to validate the DapperUserRepository.
* Should test user persistence operations like:
  + Creating a new user.
  + Fetching a user by ID, username, or email.
  + Updating user profile data.
  + Ensuring uniqueness constraints are enforced.
* Currently **empty** — no validation of repository behaviour.

**🔍 Detailed Breakdown**

* Only SPDX license header and metadata (created: 2025-06-16).
* No namespaces, classes, or test methods defined.

**⚠️ Error Handling & Validation**

* ❌ None present — critical persistence logic untested.

**🔐 Security Review**

* ✅ No secrets or PII in file.
* ❌ Missing tests mean we cannot verify that sensitive fields (like PasswordHash) are properly handled by repository.

**⚡ Performance & Reliability**

* N/A — no tests.

**📊 Observability**

* N/A — no logging or assertions.

**🧪 Testability & Coverage**

**Current coverage:** 0% for DapperUserRepository.

**Recommended test cases:**

1. **Happy path:** Insert user and retrieve by ID.
2. **Lookup:** Retrieve by Username and Email.
3. **Negative:** Duplicate username/email should fail.
4. **Update:** Change user details and confirm persistence.
5. **Edge:** Empty fields → ensure validation prevents inserts.

**🧹 Code Smells**

* **Critical:** Placeholder file with no tests.

**🔧 Refactoring Suggestions**

1. Implement tests for all CRUD operations.
   * Effort: Medium, Priority: 5.
2. Add validation tests for duplicates and null values.
   * Effort: Small, Priority: 4.
3. Add DB integration test to confirm Dapper mappings.
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* DapperUserRepository is central to persistence.
* Any schema change in Users table affects repository → tests must confirm mapping consistency.

**🗄️ Data Model Notes**

* Maps to Users table.
* Must enforce:
  + Primary key = UserId.
  + Unique indexes = Username, Email.
  + Non-null = PasswordHash.

**✅ Confidence**

**High** – File reviewed; confirmed it is empty and missing tests for critical repository functionality.

**File Review**

**File:** Tests/Infrastructure/Persistence/EndpointConfigRepositoryTests.cs  
**Layer/Type:** Tests – Infrastructure (Repository test)  
**Status:** Reviewed  
**Tokens (approx.):** ~350

**🧾 ELI5**

This file tests the EndpointConfigRepository, which retrieves endpoint configurations for Xero data sync. It ensures the repository attempts to connect to the database and handles unsupported commands gracefully.

**🎯 Purpose and Role**

* Validates **EndpointConfigRepository** behaviour.
* Ensures integration with:
  + IDbConnectionFactory → supplies database connections.
  + ILogger<EndpointConfigRepository> → logs issues.
* Confirms that unsupported operations throw exceptions as expected.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Mocks:**
  + IDbConnectionFactory → supplies DbConnection.
  + ILogger<EndpointConfigRepository>.
* **Repository under test:** EndpointConfigRepository.
* **Test case:** GetAllAsync\_Uses\_ConnectionFactory
  + Creates Mock<IDbConnection>.
  + Mocks IDbConnectionFactory.CreateConnectionAsync to return the mock connection.
  + Instantiates EndpointConfigRepository with mocks.
  + Calls GetAllAsync().
  + Uses Assert.ThrowsAnyAsync<Exception> → expects exception since repository is not fully implemented.
  + Verifies factory was called exactly once.

**⚠️ Error Handling & Validation**

* ✅ Confirms exception thrown when calling repository (method not supported).
* ❌ Missing:
  + Validation for empty/missing endpoint configs.
  + Handling of SQL errors.
  + Logging behaviour verification.

**🔐 Security Review**

* ✅ No secrets in tests.
* ❌ Does not validate safe parameterised queries or SQL injection prevention.
* ❌ No coverage for connection string sourcing (must come from Key Vault).

**⚡ Performance & Reliability**

* Lightweight, mock-driven.
* No DB integration or performance validation.
* No retries tested for transient DB failures.

**📊 Observability**

* Logger injected but not verified.
* No audit trail coverage for repository calls.

**🧪 Testability & Coverage**

**Covered:**

* Repository attempts to use IDbConnectionFactory.
* Unsupported method call throws exception.

**Missing (should add):**

1. Test successful retrieval with a fake in-memory DB.
2. Test empty DB table → returns empty list.
3. Test DB exception (e.g., timeout).
4. Verify logging occurs.

**🧹 Code Smells**

* **Medium:** Only tests exception path — no positive scenario.
* **Low:** Relies on Moq strictness for validation.
* **Info:** Doesn’t validate SQL queries.

**🔧 Refactoring Suggestions**

1. Add integration test with in-memory or test DB for GetAllAsync.
   * Effort: Medium, Priority: 5.
2. Add negative test for DB timeout/failure.
   * Effort: Small, Priority: 4.
3. Verify logger entries for observability.
   * Effort: Small, Priority: 3.
4. Ensure queries are parameterised (SQL injection safe).
   * Effort: Small, Priority: 4.

**📜 Contracts & Compatibility**

* Relies on IDbConnectionFactory.CreateConnectionAsync.
* Contract risk: if connection factory changes signature, test fails.
* Repository contract must remain stable for Xero sync to work.

**🗄️ Data Model Notes**

* EndpointConfig maps to config tables (likely cfg.xero\_endpoints).
* Schema must enforce Name, Order, and IsActive.

**✅ Confidence**

**High** – File fully reviewed; confirms repo wiring but missing positive/negative integration scenarios.

**File Review**

**File:** Tests/Infrastructure/Persistence/RawXeroPayloadRepositoryTests.cs  
**Layer/Type:** Tests – Infrastructure (Repository test)  
**Status:** Reviewed  
**Tokens (approx.):** ~420

**🧾 ELI5**

This file tests the repository that stores raw payloads from Xero. It verifies that invalid SQL identifiers cause exceptions when inserting pages or retrieving maximum page numbers.

**🎯 Purpose and Role**

* Validates **RawXeroPayloadRepository** against database schema constraints.
* Ensures invalid SQL identifiers (like table names) cause errors.
* Protects RoadmApp’s **ETL ingestion layer** from silent persistence failures.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), Moq, FluentAssertions.
* **Repository under test:** RawXeroPayloadRepository.
* **Helper:** MakeRepo() builds repository with:
  + In-memory config (connection string to local Postgres).
  + NullLogger<RawXeroPayloadRepository>.
* **Test cases:**
  + **InsertPageAsync\_rejects\_invalid\_table\_identifier\_before\_db\_connect**
    - Calls InsertPageAsync("raw.ingestion; DROP TABLE x;").
    - Expects InvalidOperationException.
    - Asserts exception message contains "Invalid SQL identifier.".
  + **GetMaxPageNumberAsync\_rejects\_invalid\_identifier\_before\_db\_connect**
    - Calls GetMaxPageNumberAsync("raw.bad name").
    - Expects InvalidOperationException.
    - Asserts exception message contains "Invalid SQL identifier.".

**⚠️ Error Handling & Validation**

* ✅ Validates identifier sanitisation at repository level.
* ❌ Missing:
  + Happy path tests (valid table name succeeds).
  + Null/empty identifier handling.
  + Actual DB integration (mock DB only).

**🔐 Security Review**

* ✅ Prevents SQL injection via identifier validation.
* ⚠️ Tests only negative paths — does not confirm parameterised queries for valid paths.
* ❌ No coverage for secret handling (connection strings must come from Key Vault in production).

**⚡ Performance & Reliability**

* Fast, lightweight negative-path checks.
* No performance/load tests for bulk insert scenarios.

**📊 Observability**

* Logger injected but unused in assertions.
* No auditability of rejected inserts.

**🧪 Testability & Coverage**

**Covered:**

* Invalid SQL identifier rejection.

**Missing (should add):**

1. Happy path insert + fetch.
2. Empty/null identifiers.
3. DB timeout/connection error handling.
4. Observability (log validation).

**🧹 Code Smells**

* **Medium:** Tests only failure paths → incomplete.
* **Low:** Hardcoded SQL injection string in test.

**🔧 Refactoring Suggestions**

1. Add **happy path test** with valid identifiers.
   * Effort: Medium, Priority: 5.
2. Add **null/empty input validation test**.
   * Effort: Small, Priority: 4.
3. Verify **logging** for observability.
   * Effort: Small, Priority: 3.
4. Add integration test with Postgres test container.
   * Effort: Medium, Priority: 3.

**📜 Contracts & Compatibility**

* Relies on contracts:
  + InsertPageAsync.
  + GetMaxPageNumberAsync.
* Contract risk: if identifiers change (schema migration), tests must be updated.

**🗄️ Data Model Notes**

* raw schema used for ingestion.
* Payload repository enforces table naming safety.
* DB should enforce:
  + tenantId foreign key.
  + Page sequence number integrity.

**✅ Confidence**

**High** – File fully reviewed; ensures SQL injection protection, but lacks happy path and full integration tests.

**File Review**

**File:** Tests/Usings.cs  
**Layer/Type:** Tests – Common (Global Usings)  
**Status:** Reviewed  
**Tokens (approx.):** ~20

**🧾 ELI5**

This file is a placeholder for global using directives in the test project. Right now, it only contains metadata comments and does not declare any global imports.

**🎯 Purpose and Role**

* Intended to centralize using statements so they don’t need to be repeated in every test file.
* Commonly used in .NET 6+ projects with implicit global using.
* Should include references like:
  + global using Xunit;
  + global using FluentAssertions;
  + global using Moq;
* Currently **empty**, so each test file must import its own dependencies.

**🔍 Detailed Breakdown**

* Only SPDX license header and metadata (created: 2025-06-16).
* No actual using statements defined.

**⚠️ Error Handling & Validation**

* N/A — no functional code.

**🔐 Security Review**

* ✅ No secrets or sensitive data.

**⚡ Performance & Reliability**

* N/A — empty file.

**📊 Observability**

* N/A — no logging.

**🧪 Testability & Coverage**

* Does not affect test coverage directly.
* If populated, could simplify test readability and maintainability.

**🧹 Code Smells**

* **Low:** File exists but unused.
* **Info:** Duplicated imports across test files could be eliminated if this file was populated.

**🔧 Refactoring Suggestions**

1. Add global imports for common test dependencies:
2. global using Xunit;
3. global using FluentAssertions;
4. global using Moq;
   * Effort: Quick Win, Priority: 5.
5. Remove unused file if project style prefers local imports.
   * Effort: Quick Win, Priority: 3.

**📜 Contracts & Compatibility**

* No external contracts — safe to modify.

**🗄️ Data Model Notes**

* Not applicable.

**✅ Confidence**

**High** – File reviewed; confirmed as empty placeholder, safe to enhance or remove.

**File Review**

**File:** Tests/Web/RateLimitingOnRejectedTests.cs  
**Layer/Type:** Tests – Web (Middleware / API Rate Limiting)  
**Status:** Reviewed  
**Tokens (approx.):** ~320

**🧾 ELI5**

This file tests that when too many requests are made too quickly, RoadmApp correctly enforces rate limiting. The first request should succeed, and the next should be rejected with status 429 Too Many Requests.

**🎯 Purpose and Role**

* Validates **ASP.NET rate limiting middleware** configuration.
* Ensures:
  + Requests exceeding quota are rejected with 429.
  + Response includes retry metadata.
  + ProblemDetails response is correctly formatted.
* Protects RoadmApp’s **API resilience** and prevents abuse.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), FluentAssertions.
* **Test harness:** Uses CustomWebApplicationFactory for test server.
* **Test case:** Second\_request\_returns\_429\_with\_retry\_after\_and\_problem\_details
  + Sends first request to /health/live → expects 200 OK.
  + Sends second request to /health/live → expects 429 Too Many Requests.
  + Asserts:
    - Retry-After header present.
    - ProblemDetails content type (application/problem+json).
    - Response JSON contains "status":429.

**⚠️ Error Handling & Validation**

* ✅ Confirms rate limiting middleware rejects excess requests.
* ❌ Missing:
  + Validation of rate limit window (e.g., how many requests per minute).
  + Behaviour after wait time passes.
  + Multi-client differentiation (per-user vs global).

**🔐 Security Review**

* ✅ Prevents API flooding (DoS mitigation).
* ❌ No test to confirm different clients are isolated by IP or auth token.
* ❌ No coverage for malicious retry logic (e.g., ignoring headers).

**⚡ Performance & Reliability**

* Lightweight, test-level HTTP client.
* No stress/load test for concurrency.
* No test for gradual ramp-up scenarios.

**📊 Observability**

* Confirms error responses follow ProblemDetails standard.
* No test for logging of rejected requests.

**🧪 Testability & Coverage**

**Covered:**

* Request rejection after quota exceeded.
* Retry header presence.
* ProblemDetails response shape.

**Missing (should add):**

1. Test reset behaviour after retry window.
2. Test multiple clients in parallel.
3. Test custom policy limits (e.g., per endpoint).
4. Test logging of rejected requests.

**🧹 Code Smells**

* **Low:** Only tests a single limit scenario.
* **Info:** Test name long and verbose.

**🔧 Refactoring Suggestions**

1. Add tests for **policy variations** (different endpoints, different quotas).
   * Effort: Medium, Priority: 5.
2. Add **time-based reset tests**.
   * Effort: Medium, Priority: 4.
3. Verify logging occurs when requests are rejected.
   * Effort: Small, Priority: 3.

**📜 Contracts & Compatibility**

* Relies on ASP.NET Core rate limiting middleware.
* If middleware or policy config changes, test expectations must be updated.

**🗄️ Data Model Notes**

* N/A (web middleware, not persistence).

**✅ Confidence**

**High** – File fully reviewed; strong test for rejection path, but missing coverage for reset and multi-client scenarios.

**File Review**

**File:** Tests/Web/RateLimitingTests.cs  
**Layer/Type:** Tests – Web (Middleware / API Rate Limiting)  
**Status:** Reviewed  
**Tokens (approx.):** ~250

**🧾 ELI5**

This file tests RoadmApp’s rate limiting policy. It verifies that the first request to an endpoint succeeds, and that the second one is blocked with 429 Too Many Requests.

**🎯 Purpose and Role**

* Validates **ASP.NET Core rate limiting middleware** applied to web endpoints.
* Ensures that excessive API calls are rejected.
* Confirms RoadmApp’s **resilience and anti-abuse mechanisms** are configured correctly.

**🔍 Detailed Breakdown**

* **Frameworks:** xUnit ([Fact]), FluentAssertions.
* **Test harness:** Uses CustomWebApplicationFactory.
* **Test case:** Second\_request\_is\_limited\_to\_429
  + Creates HttpClient from factory.
  + Sends first request to /health/live → expects 200 OK.
  + Sends second request to /health/live → expects 429 Too Many Requests.

**⚠️ Error Handling & Validation**

* ✅ Confirms quota enforcement.
* ❌ Missing:
  + Retry-after behaviour.
  + ProblemDetails validation.
  + Multi-client differentiation (per-user/IP).
  + Reset window handling.

**🔐 Security Review**

* ✅ Confirms DoS prevention at API level.
* ❌ No test to confirm different clients are not rate-limited globally.

**⚡ Performance & Reliability**

* Lightweight, runs two HTTP requests.
* No concurrency load simulation.
* No test for higher request volumes.

**📊 Observability**

* Only checks HTTP status codes.
* No logging verification for rejected requests.

**🧪 Testability & Coverage**

**Covered:**

* Request blocking after quota exceeded.

**Missing (should add):**

1. Check Retry-After header.
2. Validate ProblemDetails JSON structure.
3. Test multi-client behaviour.
4. Test quota reset after wait window.
5. Stress test with concurrent requests.

**🧹 Code Smells**

* **Low:** Minimal test → doesn’t cover full policy.
* **Info:** Duplicates coverage from RateLimitingOnRejectedTests (overlap).

**🔧 Refactoring Suggestions**

1. Merge with RateLimitingOnRejectedTests to avoid duplication.
   * Effort: Small, Priority: 3.
2. Add retry-after and ProblemDetails assertions.
   * Effort: Small, Priority: 5.
3. Add parallel client simulation.
   * Effort: Medium, Priority: 4.

**📜 Contracts & Compatibility**

* Relies on ASP.NET Core middleware contracts.
* Test expectations tied to configured quota policy.
* If policy changes, test must be updated.

**🗄️ Data Model Notes**

* N/A (web middleware).

**✅ Confidence**

**High** – File fully reviewed; confirms basic rate limiting works, but lacks depth and overlaps with other rate limit test.