# 1.Why choose this Core feature:

This feature was chosen for comprehensive testing because it is a **core, complex, and high-risk** component of the application. Its reliability is critical to the API's main purpose.

* **Core Functionality:** The documentation identifies it as a "**core component**" of the TestCase Generator API. Its primary function is to generate the main deliverable (a use case report).
* **Complex Integration:** The feature is not a simple function. It's an entire workflow that chains multiple services together:
  1. API Endpoint receives an HTTP request.
  2. GeminiService generates a prompt.
  3. An external call is made to the **Google Gemini API**.
  4. The AI's text response is **parsed and validated**.
  5. ExcelService transforms the parsed data into an **Excel file** (byte[]).
  6. The API returns this file to the user. A failure at any link in this chain will cause the entire feature to fail.
* **High Risk (External Dependencies):** The feature depends on the external Google Gemini API. This introduces risks that must be tested, such as network failures, API timeouts, and receiving **malformed JSON** or unexpected data from the AI.
* **Complex Data Transformation:** The feature's main job is to transform a simple text input (like "User Login") into a highly structured UseCaseReport object, which is then transformed again into a complex Excel spreadsheet. Testing is required to ensure this data parsing and file generation is accurate, especially with edge cases like empty lists or very long text.

# 

# **2.Analyze code with AI:**

1. GenerateUseCaseReportAsync(useCaseName, additionalContext)

Main: Generate comprehensive use case report using AI

Inputs:

useCaseName (string) - Name of the use case

additionalContext (string, optional) - Additional context for the use case

Returns: UseCaseReport object with complete use case specification

Edge cases:

Empty use case name

Null additional context

Very long use case names

Special characters in use case name

Dependencies:

Google Gemini API

HTTP client configuration

JSON parsing and validation

2. CreateUseCaseReportPrompt(useCaseName, additionalContext)

Main: Generate AI prompt for use case report generation

Inputs:

useCaseName (string) - Use case name

additionalContext (string, optional) - Additional context

Returns: string - Formatted prompt for AI

Edge cases:

Null or empty use case name

Very long context strings

Special characters and formatting

Dependencies:

String formatting

Template generation

3. ParseUseCaseReportResponse(generatedText)

Main: Parse AI response into UseCaseReport object

Inputs:

generatedText (string) - Raw AI response text

Returns: UseCaseReport object

Edge cases:

Malformed JSON response

Missing required fields

Invalid data types

Empty or null response

Dependencies:

JSON deserialization

Data validation

Error handling

4. ExportUseCaseReportToExcel(useCaseReport)

Main: Export use case report to Excel format

Inputs:

useCaseReport (UseCaseReport) - Complete use case data

Returns: byte[] - Excel file bytes

Edge cases:

Null use case report

Empty collections (preconditions, postconditions, flows)

Very long text fields

Missing required fields

Dependencies:

ClosedXML library

Excel formatting

Memory stream handling

5. GenerateUseCaseReport(UseCaseTableRequest)

Main: Controller endpoint for use case report generation

Inputs:

request (UseCaseTableRequest) - HTTP request body

Returns: IActionResult - Excel file download or error response

Edge cases:

Invalid request body

Missing required fields

AI service failures

Excel generation errors

Dependencies:

GeminiService

ExcelService

HTTP response handling

# 3.Design test case:

prompt: give me cmd to create unittest with the folder testcasegenrator.test inside the sln and flow to mitigate. Include the package that suitable for testing for .net 9.0

# Create test projects directory

mkdir TestcaseGenerator.Tests

cd TestcaseGenerator.Tests

# Create main test project

dotnet new xunit -n TestcaseGenerator.Tests

cd TestcaseGenerator.Tests

# Add project reference to main project

dotnet add reference "..\..\TestcaseGenerator\TestcaseGenerator.csproj"

# Add required NuGet packages

dotnet add package FluentAssertions

dotnet add package Moq

dotnet add package Microsoft.AspNetCore.Mvc.Testing

# Create service test project

cd ..

dotnet new xunit -n TestcaseGenerator.Service.Tests

cd TestcaseGenerator.Service.Tests

# Add project reference to main project

dotnet add reference "..\..\TestcaseGenerator\TestcaseGenerator.csproj"

# Add required NuGet packages

dotnet add package FluentAssertions

dotnet add package Moq

# Create integration test project

cd ..

dotnet new xunit -n TestcaseGenerator.Integration.Tests

cd TestcaseGenerator.Integration.Tests

# Add project reference to main project

dotnet add reference "..\..\TestcaseGenerator\TestcaseGenerator.csproj"

# Add required NuGet packages

dotnet add package FluentAssertions

dotnet add package Moq

dotnet add package Microsoft.AspNetCore.Mvc.Testing

```

**#### Add Projects to Main Solution**

```bash

# Navigate back to main directory

cd "C:\AI FOR SE\BE"

# Add test projects to solution

dotnet sln add TestcaseGenerator.Tests\TestcaseGenerator.Tests\TestcaseGenerator.Tests.csproj

dotnet sln add TestcaseGenerator.Tests\TestcaseGenerator.Service.Tests\TestcaseGenerator.Service.Tests.csproj

dotnet sln add TestcaseGenerator.Tests\TestcaseGenerator.Integration.Tests\TestcaseGenerator.Integration.Tests.csproj

# Verify solution structure

dotnet sln list

```

## Result:

## Unit:

## ### 1. Controller Tests - TestcaseControllerUseCaseReportTests.cs

**\*\*Purpose\*\*: Test the API controller endpoint for use case report generation**

**\*\*Dependencies\*\*: None (Pure unit tests)**

**\*\*Priority\*\*: HIGH**

**#### Happy Path Tests**

**| Test Case | Input | Expected | Dependencies |**

**|-----------|-------|----------|--------------|**

**| `GenerateUseCaseReport\_ValidRequest\_ReturnsExcelFile` | Valid UseCaseTableRequest | Excel file response | None |**

**| `GenerateUseCaseReport\_ValidRequestWithoutContext\_ReturnsExcelFile` | Request with null context | Excel file response | None |**

**#### Input Validation Tests**

**| Test Case | Input | Expected | Dependencies |**

**|-----------|-------|----------|--------------|**

**| `GenerateUseCaseReport\_EmptyUseCaseName\_ReturnsError` | Empty use case name | Error response | None |**

**| `GenerateUseCaseReport\_NullUseCaseName\_ReturnsError` | Null use case name | Error response | None |**

**| `GenerateUseCaseReport\_WhitespaceUseCaseName\_ReturnsError` | Whitespace use case name | Error response | None |**

**| `GenerateUseCaseReport\_VeryLongUseCaseName\_HandlesGracefully` | 1000+ character name | Handles gracefully | None |**

**| `GenerateUseCaseReport\_SpecialCharactersInUseCaseName\_HandlesGracefully` | Special characters | Handles gracefully | None |**

**#### Request Validation Tests**

**| Test Case | Input | Expected | Dependencies |**

**|-----------|-------|----------|--------------|**

**| `GenerateUseCaseReport\_WithNullRequest\_ReturnsError` | Null request body | Error response | None |**

**| `GenerateUseCaseReport\_WithValidRequestButNoContext\_HandlesCorrectly` | Request without context | Handles correctly | None |**

**| `GenerateUseCaseReport\_WithEmptyContext\_HandlesCorrectly` | Empty context string | Handles correctly | None |**

**#### Configuration Tests**

**| Test Case | Input | Expected | Dependencies |**

**|-----------|-------|----------|--------------|**

**| `TestcaseController\_WithValidConfiguration\_InitializesCorrectly` | Valid configuration | Controller initialized | None |**

**| `TestcaseController\_WithNullHttpClient\_ThrowsException` | Null HttpClient | ArgumentNullException | None |**

**| `TestcaseController\_WithNullConfiguration\_ThrowsException` | Null configuration | NullReferenceException | None |**

**#### Edge Case Tests**

**| Test Case | Input | Expected | Dependencies |**

**|-----------|-------|----------|--------------|**

**| `GenerateUseCaseReport\_WithVeryLongContext\_HandlesCorrectly` | 10000+ character context | Handles correctly | None |**

**| `GenerateUseCaseReport\_WithSpecialCharactersInContext\_HandlesCorrectly` | Special characters in context | Handles correctly | None |**

### ### 2. Service Tests - GeminiServiceUseCaseReportTests.cs

**\*\*Purpose\*\***: Test the AI service for use case report generation

**\*\*Dependencies\*\***: None (Pure unit tests)

**\*\*Priority\*\***: HIGH

**#### Happy Path Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GenerateUseCaseReportAsync\_ValidRequest\_WillAttemptProcessing` | Valid use case name | Attempts processing | None |

| `GenerateUseCaseReportAsync\_ValidRequestWithoutContext\_WillAttemptProcessing` | Request without context | Attempts processing | None |

**#### Input Validation Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GenerateUseCaseReportAsync\_WithNullUseCaseName\_ThrowsException` | Null use case name | ArgumentNullException | None |

| `GenerateUseCaseReportAsync\_WithEmptyUseCaseName\_ThrowsException` | Empty use case name | ArgumentException | None |

| `GenerateUseCaseReportAsync\_WithWhitespaceUseCaseName\_ThrowsException` | Whitespace use case name | ArgumentException | None |

**#### Edge Case Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GenerateUseCaseReportAsync\_WithVeryLongUseCaseName\_HandlesGracefully` | 1000+ character name | Handles gracefully | None |

| `GenerateUseCaseReportAsync\_WithSpecialCharacters\_HandlesGracefully` | Special characters | Handles gracefully | None |

| `GenerateUseCaseReportAsync\_WithVeryLongContext\_HandlesGracefully` | 10000+ character context | Handles gracefully | None |

**#### Configuration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GeminiService\_WithValidConfiguration\_InitializesCorrectly` | Valid configuration | Service initialized | None |

| `GeminiService\_WithNullApiKey\_ThrowsException` | Null API key | InvalidOperationException | None |

| `GeminiService\_WithEmptyApiKey\_ThrowsException` | Empty API key | InvalidOperationException | None |

| `GeminiService\_WithNullBaseUrl\_UsesDefaultUrl` | Null base URL | Uses default URL | None |

**#### Service Initialization Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GeminiService\_WithValidHttpClient\_InitializesCorrectly` | Valid HttpClient | Service initialized | None |

| `GeminiService\_WithNullHttpClient\_ThrowsException` | Null HttpClient | ArgumentNullException | None |

| `GeminiService\_WithNullConfiguration\_ThrowsException` | Null configuration | ArgumentNullException | None |

**#### Error Handling Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `GenerateUseCaseReportAsync\_WithInvalidApiKey\_ThrowsException` | Invalid API key | Exception thrown | None |

| `GenerateUseCaseReportAsync\_WithInvalidBaseUrl\_ThrowsException` | Invalid base URL | Exception thrown | None |

### **### 3. Service Tests - ExcelServiceUseCaseReportTests.cs**

**\*\*Purpose\*\***: Test the Excel generation service for use case reports

**\*\*Dependencies\*\***: None (Pure unit tests)

**\*\*Priority\*\***: HIGH

**#### Happy Path Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ReturnsExcelBytes` | Valid UseCaseReport | Excel bytes array | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsCorrectHeader` | Valid report | Contains header | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsBasicInformation` | Valid report | Contains basic info | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsPreconditions` | Valid report | Contains preconditions | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsPostconditions` | Valid report | Contains postconditions | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsNormalFlow` | Valid report | Contains normal flow | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsAlternativeFlows` | Valid report | Contains alternative flows | None |

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_ContainsExceptions` | Valid report | Contains exceptions | None |

**#### Edge Case Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `ExportUseCaseReportToExcel\_NullUseCaseReport\_ThrowsException` | Null report | ArgumentNullException | None |

| `ExportUseCaseReportToExcel\_UseCaseReportWithEmptyCollections\_HandlesCorrectly` | Empty collections | Handles correctly | None |

| `ExportUseCaseReportToExcel\_UseCaseReportWithVeryLongText\_HandlesCorrectly` | 10000+ character text | Handles correctly | None |

| `ExportUseCaseReportToExcel\_UseCaseReportWithSpecialCharacters\_HandlesCorrectly` | Special characters | Handles correctly | None |

**#### Data Validation Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `ExportUseCaseReportToExcel\_UseCaseReportWithMultipleItems\_ContainsAllItems` | Complex report | Contains all items | None |

| `ExportUseCaseReportToExcel\_UseCaseReportWithMultipleFlows\_ContainsAllFlows` | Multiple flows | Contains all flows | None |

**#### Excel Formatting Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `ExportUseCaseReportToExcel\_ValidUseCaseReport\_HasProperFormatting` | Valid report | Proper formatting | None |

### ### 4. Model Tests - [UseCaseReportModelTests.cs](http://usecasereportmodeltests.cs)

**\*\*Purpose\*\***: Test data models and serialization

**\*\*Dependencies\*\***: None (Pure unit tests)

**\*\*Priority\*\***: HIGH

**#### UseCaseReport Model Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_DefaultConstructor\_InitializesCorrectly` | Default constructor | All properties initialized | None |

| `UseCaseReport\_WithValidData\_PropertiesSetCorrectly` | Valid data | Properties set correctly | None |

| `UseCaseReport\_WithCollections\_CollectionsSetCorrectly` | Collections data | Collections set correctly | None |

| `UseCaseReport\_JsonSerialization\_WorksCorrectly` | Valid report | JSON serialization works | None |

| `UseCaseReport\_JsonDeserializationWithNullValues\_WorksCorrectly` | JSON with nulls | Deserialization works | None |

**#### FlowStep Model Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `FlowStep\_DefaultConstructor\_InitializesCorrectly` | Default constructor | Properties initialized | None |

| `FlowStep\_WithValidData\_PropertiesSetCorrectly` | Valid data | Properties set correctly | None |

| `FlowStep\_JsonSerialization\_WorksCorrectly` | Valid step | JSON serialization works | None |

**#### AlternativeFlow Model Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `AlternativeFlow\_DefaultConstructor\_InitializesCorrectly` | Default constructor | Properties initialized | None |

| `AlternativeFlow\_WithValidData\_PropertiesSetCorrectly` | Valid data | Properties set correctly | None |

| `AlternativeFlow\_JsonSerialization\_WorksCorrectly` | Valid flow | JSON serialization works | None |

**#### ExceptionFlow Model Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `ExceptionFlow\_DefaultConstructor\_InitializesCorrectly` | Default constructor | Properties initialized | None |

| `ExceptionFlow\_WithValidData\_PropertiesSetCorrectly` | Valid data | Properties set correctly | None |

| `ExceptionFlow\_JsonSerialization\_WorksCorrectly` | Valid exception | JSON serialization works | None |

**#### Edge Case Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_WithVeryLongText\_HandlesCorrectly` | 10000+ character text | Handles correctly | None |

| `UseCaseReport\_WithSpecialCharacters\_HandlesCorrectly` | Special characters | Handles correctly | None |

| `UseCaseReport\_WithEmptyCollections\_HandlesCorrectly` | Empty collections | Handles correctly | None |

## Integration:

### ### 5. Integration Tests - UseCaseReportIntegrationTests.cs

**\*\*Purpose\*\***: Test end-to-end workflows and data flow

**\*\*Dependencies\*\***: External services (AI, Excel generation)

**\*\*Priority\*\***: MEDIUM

**#### Model Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_ModelSerialization\_WorksCorrectly` | Valid report | Serialization works | None |

| `UseCaseTableRequest\_ModelSerialization\_WorksCorrectly` | Valid request | Serialization works | None |

**#### Data Validation Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_WithComplexData\_SerializesCorrectly` | Complex report | Serialization works | None |

| `UseCaseReport\_WithEmptyCollections\_HandlesCorrectly` | Empty collections | Handles correctly | None |

**#### Edge Case Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_WithVeryLongText\_HandlesCorrectly` | 10000+ character text | Handles correctly | None |

| `UseCaseReport\_WithSpecialCharacters\_HandlesCorrectly` | Special characters | Handles correctly | None |

**#### Flow Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `FlowStep\_Serialization\_WorksCorrectly` | Valid step | Serialization works | None |

| `AlternativeFlow\_Serialization\_WorksCorrectly` | Valid flow | Serialization works | None |

| `ExceptionFlow\_Serialization\_WorksCorrectly` | Valid exception | Serialization works | None |

**#### End-to-End Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_CompleteWorkflow\_DataFlowWorksCorrectly` | Complete workflow | Data flow works | None |

| `UseCaseReport\_WithMultipleFlows\_ComplexDataHandlesCorrectly` | Complex data | Handles correctly | None |

**#### Performance Integration Tests**

| Test Case | Input | Expected | Dependencies |

|-----------|-------|----------|--------------|

| `UseCaseReport\_LargeDataSet\_PerformanceIsAcceptable` | Large dataset | Performance acceptable | None |

# 4. Generate test code:

## Gemini Service:

You are a C# Unit Testing Expert using xUnit, Moq, and FluentAssertions.

I have a GeminiService.cs class that needs better unit testing. I also have an *existing* test file, GeminiServiceUseCaseReportTests.cs, but it has two major problems:

1. It only tests the GenerateUseCaseReportAsync method and the constructor. It completely misses GenerateTestCasesAsync and GenerateUseCaseTableAsync.
2. The existing tests for GenerateUseCaseReportAsync are flawed. They use new HttpClient() and just Assert.ThrowsAsync<Exception>. This doesn't actually test the method's logic (like success, API failure, or parsing).

Your task is to generate a new, complete test file named GeminiServiceTests.cs that fixes these problems and provides comprehensive coverage.

**Requirements:**

1. **Mock HttpClient:** You MUST NOT use new HttpClient(). You must mock the HttpClient's behavior by using a Mock<HttpMessageHandler>. This is critical for simulating API responses.
2. **Cover All Public Methods:** Write tests for GenerateTestCasesAsync, GenerateUseCaseTableAsync, and *fix* the tests for GenerateUseCaseReportAsync.
3. **Use IConfiguration Mock:** Continue using Mock<IConfiguration> as shown in the existing tests.
4. **Test Scenarios:** For *each* of the three Generate...Async methods, you must generate tests for the following scenarios:
   * **Success (200 OK):**
     + Simulate a HttpStatusCode.OK response with a valid, well-formed JSON payload (based on the GeminiApiResponse structure).
     + Assert that the method returns the correctly parsed object (UseCaseReport or GeminiTestCaseResponse).
     + Assert that Success is true for GeminiTestCaseResponse.
   * **API Failure (e.g., 400 Bad Request):**
     + Simulate a HttpStatusCode.BadRequest or HttpStatusCode.InternalServerError response.
     + Assert that the method throws an Exception containing the error message from the API.
   * **Malformed JSON Response:**
     + Simulate a HttpStatusCode.OK response but with invalid JSON (e.g., "{"bad": "json"").
     + Assert that the method throws an Exception (from the Parse... methods) indicating a parsing error.
   * **Empty Candidates Response:**
     + For GenerateTestCasesAsync and GenerateUseCaseTableAsync, simulate a HttpStatusCode.OK response with valid JSON but an empty candidates array.
     + Assert that the method returns a GeminiTestCaseResponse where Success is false and a proper Message is set.
5. **Input Validation:** The current GeminiService code does not have explicit null/empty checks for the string inputs in its public methods. Create tests that check for ArgumentNullException or ArgumentException for userRequirement and useCaseName. (This will require adding the validation to the GeminiService implementation, but you should write the tests as if it exists).

## 

## Excel Service:

You are a C# Unit Testing expert using xUnit, FluentAssertions, and the ClosedXML.Excel library.

Your task is to create a unit test class named ExcelServiceUseCaseReportTests for the provided ExcelService.cs class.

**Base Code (ExcelService.cs):**

**Instructions for the Test Class:**

1. **Test Focus:** The test class must **only** contain tests for the ExportUseCaseReportToExcel(UseCaseReport report) method. Do *not* generate tests for ExportToExcel(TestCaseRequest request) or any of the private helper methods.
2. **Instantiation:** The ExcelService must be instantiated directly in the test class's constructor (\_excelService = new ExcelService();). Do not use any mocks.
3. **Test Scenarios (Must Include):**
   * **Happy Path:**
     + ...\_ReturnsExcelBytes: Checks that the byte[] result is not null or empty.
     + ...\_ContainsCorrectHeader: Checks that the generated Excel file's cell A1 contains the "II. Requirement Specifications" title.
     + ...\_ContainsBasicInformation: Checks for basic fields like "Created By:" and its value.
     + ...\_ContainsPreconditions: Checks that precondition text is present.
     + ...\_ContainsPostconditions: Checks that postcondition text is present.
     + ...\_ContainsNormalFlow: Checks that normal flow steps are present.
     + ...\_ContainsAlternativeFlows: Checks that alternative flow steps are present.
     + ...\_ContainsExceptions: Checks that exception flow text is present.
   * **Edge Cases:**
     + ...\_NullUseCaseReport\_ThrowsException: Asserts that passing a null report throws an ArgumentNullException.
     + ...\_UseCaseReportWithEmptyCollections\_HandlesCorrectly: Confirms the method runs successfully when all List<> properties on the report are empty.
     + ...\_UseCaseReportWithVeryLongText\_HandlesCorrectly: Confirms the method runs successfully when report properties contain very long strings (e.g., 10,000+ characters).
     + ...\_UseCaseReportWithSpecialCharacters\_HandlesCorrectly: Confirms the method runs successfully when report properties contain special characters.
   * **Data Validation:**
     + ...\_UseCaseReportWithMultipleItems\_ContainsAllItems: Confirms that when a report has *multiple* preconditions, *all* of them are present in the file.
     + ...\_UseCaseReportWithMultipleFlows\_ContainsAllFlows: Confirms that *all* normal flow steps are present.
   * **Formatting:**
     + ...\_HasProperFormatting: A simple test to check that the worksheet's RangeUsed() is not null.
4. **Test Helper Methods (Crucial):** The test class *must* include the following private helper methods to support the tests:
   * CreateValidUseCaseReport(): A helper that returns a standard, fully-populated UseCaseReport object with sample data for all fields.
   * CreateComplexUseCaseReport(): A helper that returns a report with *multiple* items in its collections (e.g., multiple preconditions, postconditions, flows) to test data validation.
   * FindRowContaining(IXLWorksheet worksheet, string text): A helper method that uses ClosedXML to iterate through the worksheet and find the row number containing a specific string.
   * GetWorksheetContent(IXLWorksheet worksheet): A helper method that reads all cell values from the worksheet into a single string.

Generate the complete ExcelServiceUseCaseReportTests.cs file, including all necessary using statements, the test class, the constructor, and all the tests and helper methods described.

## Models:

You are a C# Unit Testing expert using xUnit, FluentAssertions, and System.Text.Json.

Your task is to create a unit test class named UseCaseReportModelTests for the provided TestcaseGenerator.Models.UseCaseReport.cs file.

**Base Code (**[**UseCaseReport.cs**](http://usecasereport.cs)**):**

**Instructions for the Test Class:**

1. **Test All Models: You must generate tests for all four model classes: UseCaseReport, FlowStep, AlternativeFlow, and ExceptionFlow.**
2. **Test Class Structure: The test class UseCaseReportModelTests must be organized using the following regions:**
   * **#region UseCaseReport Model Tests**
   * **#region FlowStep Model Tests**
   * **#region AlternativeFlow Model Tests**
   * **#region ExceptionFlow Model Tests**
   * **#region Edge Case Tests**
   * **#region Helper Methods**
3. **No Mocks: These are POCO model tests. Do not use any mocks.**
4. **Test Scenarios:**
   * **For *each* of the 4 model classes:**
     + **...\_DefaultConstructor\_InitializesCorrectly: Test that the parameterless constructor initializes all string properties to "" and all List properties to a non-null, empty list.**
     + **...\_WithValidData\_PropertiesSetCorrectly: Test that properties can be set and read correctly.**
     + **...\_JsonSerialization\_WorksCorrectly: Test that the object can be serialized and then deserialized back into an object with the same property values (a round-trip test).**
   * **For UseCaseReport specifically:**
     + **...\_WithCollections\_CollectionsSetCorrectly: Test that items added to the List properties are stored correctly and the counts are right.**
     + **...\_JsonDeserializationWithNullValues\_WorksCorrectly: Create a specific test that uses a raw JSON string ("""...""") where string properties like secondaryActors, trigger, and description are set to null. Then, deserialize this string and assert that those properties on the resulting object are null.**
   * **For Edge Case Tests Region:**
     + **...\_WithVeryLongText\_HandlesCorrectly: Test serializing/deserializing a UseCaseReport with 10,000+ character strings.**
     + **...\_WithSpecialCharacters\_HandlesCorrectly: Test serializing/deserializing a UseCaseReport with various special characters.**
     + **...\_WithEmptyCollections\_HandlesCorrectly: Test serializing/deserializing a UseCaseReport where all List properties are new().**
5. **Helper Methods:**
   * **The class *must* include a private helper method CreateValidUseCaseReport() inside the Helper Methods region. This method should return a fully populated UseCaseReport object with sample data for all properties and collections, to be reused by other tests.**

**Generate the complete UseCaseReportModelTests.cs file, including all necessary using statements, the test class, and all the tests and helper methods as described.**

## Controller:

You are a C# Unit Testing expert using xUnit, Moq, and FluentAssertions.

Your task is to create a unit test class named TestcaseControllerUseCaseReportTests for the provided TestcaseController.cs class.

**Base Code (TestcaseController.cs):**

**Instructions for the Test Class:**

1. **Test Focus:** The test class must **only** contain tests for the TestcaseController constructor and the GenerateUseCaseReport endpoint. Do *not* generate tests for GenerateAndDownload or GenerateUseCaseTable.
2. **HttpClient Strategy (Very Important):**
   * You **must not** mock the HttpClient.
   * In the test class's constructor, you must instantiate a real HttpClient using \_httpClient = new HttpClient();.
3. **Assertion Strategy (Very Important):**
   * Because a *real* HttpClient is used, the call to \_geminiService.GenerateUseCaseReportAsync will always fail (e.g., network error).
   * The controller's try/catch block will catch this Exception and return StatusCode(500, ...).
   * Therefore, all tests that call the GenerateUseCaseReport endpoint **must assert that the result is of type ObjectResult**. Do *not* assert FileResult.
4. **Mocking:**
   * You must mock IConfiguration.
   * Set up the mock to return "test-api-key" for "GeminiApi:ApiKey" and "https://test-api.com" for "GeminiApi:BaseUrl".
5. **Test Class Structure:** The test class must be organized using the following regions:
   * #region GenerateUseCaseReport - Happy Path Tests
   * #region GenerateUseCaseReport - Input Validation Tests
   * #region GenerateUseCaseReport - Request Validation Tests
   * #region Controller Configuration Tests
   * #region Edge Case Tests
6. **Test Scenarios to Include:**
   * **Controller Configuration:**
     + ...\_WithValidConfiguration\_InitializesCorrectly
     + ...\_WithNullHttpClient\_ThrowsException: (Asserts ArgumentNullException)
     + ...\_WithNullConfiguration\_ThrowsException: (Asserts NullReferenceException)
   * **Happy Path:**
     + ...\_ValidRequest\_ReturnsExcelFile: (Asserts ObjectResult)
     + ...\_ValidRequestWithoutContext\_ReturnsExcelFile: (Asserts ObjectResult)
   * **Input Validation:**
     + ...\_EmptyUseCaseName\_ReturnsError: (Asserts ObjectResult)
     + ...\_NullUseCaseName\_ReturnsError: (Asserts ObjectResult)
     + ...\_WhitespaceUseCaseName\_ReturnsError: (Asserts ObjectResult)
   * **Request Validation:**
     + ...\_WithNullRequest\_ReturnsError: (Asserts ObjectResult)
   * **Edge Cases:**
     + ...\_VeryLongUseCaseName\_HandlesGracefully: (Asserts ObjectResult)
     + ...\_SpecialCharactersInUseCaseName\_HandlesGracefully: (Asserts ObjectResult)
     + ...\_WithVeryLongContext\_HandlesCorrectly: (Asserts ObjectResult)
     + ...\_WithSpecialCharactersInContext\_HandlesCorrectly: (Asserts ObjectResult)

Generate the complete TestcaseControllerUseCaseReportTests.cs file, including all necessary using statements, the test class, the constructor setup, and all the tests and regions as described.

## Integration:

You are a C# Unit Testing expert using xUnit, FluentAssertions, and System.Text.Json.

Your task is to create an "integration" test class named UseCaseReportIntegrationTests that focuses *exclusively* on testing the JSON serialization and deserialization of the UseCaseReport models.

**Assumed Models:** You can assume the following models exist in the TestcaseGenerator.Models namespace: UseCaseReport, FlowStep, AlternativeFlow, ExceptionFlow, and UseCaseTableRequest.

**Instructions for the Test Class:**

1. **Class Name & Namespace:** The class must be named UseCaseReportIntegrationTests and be in the TestcaseGenerator.Integration.Tests namespace.
2. **No Mocks:** This class must not use any mocks. It should only instantiate model objects and use System.Text.Json.JsonSerializer.
3. **Test Class Structure:** The class must be organized into the following seven regions, in this order:
   * #region Model Integration Tests
   * #region Data Validation Integration Tests
   * #region Edge Case Integration Tests
   * #region Flow Integration Tests
   * #region End-to-End Integration Tests
   * #region Performance Integration Tests
   * #region Helper Methods
4. **Helper Methods (Must be implemented in the Helper Methods region):**
   * CreateMockUseCaseReport(): Creates a standard, valid UseCaseReport with 2 preconditions, 2 postconditions, 2 normal flow steps, 1 alternative flow, and 1 exception.
   * CreateComplexUseCaseReport(): Creates a more complex report with 3 preconditions, 3 postconditions, 5 normal flow steps, 2 alternative flows, and 2 exceptions.
   * CreateLargeUseCaseReport(): Creates a large report for performance testing. It must use a for loop to add 100 preconditions, 100 postconditions, 100 normal flow steps, and another for loop to add 50 alternative flows and 50 exceptions.
5. **Test Scenarios to Include:**
   * **Model Integration:**
     + ...\_ModelSerialization\_WorksCorrectly: Tests a standard serialize/deserialize round-trip for UseCaseReport.
     + ...\_ModelSerialization\_WorksCorrectly: Tests a standard serialize/deserialize round-trip for UseCaseTableRequest.
   * **Data Validation:**
     + ...\_WithComplexData...: Serializes/deserializes the CreateComplexUseCaseReport and asserts collection counts.
     + ...\_WithEmptyCollections...: Serializes/deserializes a UseCaseReport with all List<> properties initialized but empty.
   * **Edge Cases:**
     + ...\_WithVeryLongText...: Serializes/deserializes a report with 10,000+ character strings in its properties.
     + ...\_WithSpecialCharacters...: Serializes/deserializes a report with special characters in its string properties.
   * **Flow Integration:**
     + FlowStep\_Serialization\_WorksCorrectly
     + AlternativeFlow\_Serialization\_WorksCorrectly
     + ExceptionFlow\_Serialization\_WorksCorrectly
   * **End-to-End:**
     + ...\_CompleteWorkflow...: A test that serializes/deserializes *both* a UseCaseTableRequest and a UseCaseReport in the same test.
     + ...\_WithMultipleFlows...: A test that serializes/deserializes the CreateComplexUseCaseReport and asserts the collection counts (similar to the Data Validation test).
   * **Performance:**
     + ...\_LargeDataSet...: A test that serializes/deserializes the CreateLargeUseCaseReport, measures the time, and asserts the TotalMilliseconds is less than 1000.

Generate the complete UseCaseReportIntegrationTests.cs file, including all using statements, the test class, all regions, all tests, and all helper methods as described.

# 5.Run test

prompt: give me cmd to test with specific one and development cmd. Beside that adding the debugging cmd to catch up with the testing  
**### 3. Test Execution Commands**

**#### Run All Tests**

```bash

# Run all tests in solution

dotnet test

# Run tests with detailed output

dotnet test --verbosity normal

# Run tests with logger output

dotnet test --logger "console;verbosity=detailed"

```

**#### Run Specific Test Projects**

```bash

# Run controller tests only

dotnet test TestcaseGenerator.Tests\TestcaseGenerator.Tests\TestcaseGenerator.Tests.csproj

# Run service tests only

dotnet test TestcaseGenerator.Tests\TestcaseGenerator.Service.Tests\TestcaseGenerator.Service.Tests.csproj

# Run integration tests only

dotnet test TestcaseGenerator.Tests\TestcaseGenerator.Integration.Tests\TestcaseGenerator.Integration.Tests.csproj

```

**#### Run Tests by Category**

```bash

# Run unit tests (controller and service tests)

dotnet test --filter "TestcaseController|GeminiService|ExcelService|UseCaseReport"

# Run integration tests

dotnet test --filter "Integration"

# Run specific test class

dotnet test --filter "TestcaseControllerUseCaseReportTests"

dotnet test --filter "GeminiServiceUseCaseReportTests"

dotnet test --filter "ExcelServiceUseCaseReportTests"

```

**#### Run Tests with Coverage**

```bash

# Install coverage tool

dotnet tool install -g dotnet-reportgenerator-globaltool

# Run tests with coverage

dotnet test --collect:"XPlat Code Coverage"

# Generate coverage report

reportgenerator -reports:"\*\*/coverage.cobertura.xml" -targetdir:"coverage" -reporttypes:"Html"

```

**### 4. Development Commands**

**#### Create New Test Files**

```bash

# Create new test file

dotnet new class -n NewTestClass

# Add test file to project

dotnet add TestcaseGenerator.Tests\TestcaseGenerator.Tests\TestcaseGenerator.Tests.csproj file NewTestClass.cs

```

**#### Clean and Rebuild**

```bash

# Clean solution

dotnet clean

# Clean and rebuild

dotnet clean && dotnet build

# Clean, restore, and build

dotnet clean && dotnet restore && dotnet build

```

**### 5. Debugging Commands**

**#### Run Tests in Debug Mode**

```bash

# Run tests with debug output

dotnet test --logger "console;verbosity=diagnostic"

# Run specific test with debug

dotnet test --filter "TestcaseControllerUseCaseReportTests" --logger "console;verbosity=diagnostic"

```

**#### Run Tests with Specific Configuration**

```bash

# Run tests with specific configuration

dotnet test --configuration Debug

dotnet test --configuration Release

# Run tests with specific framework

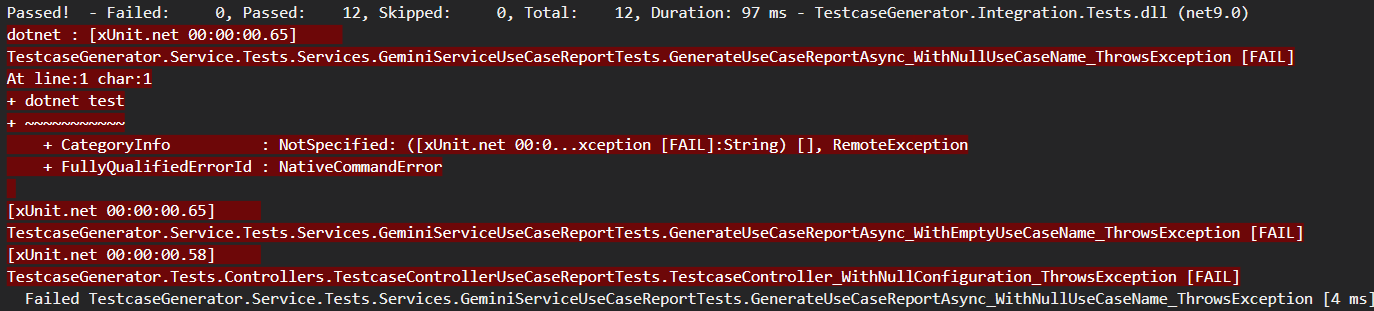
dotnet test --framework net9.0

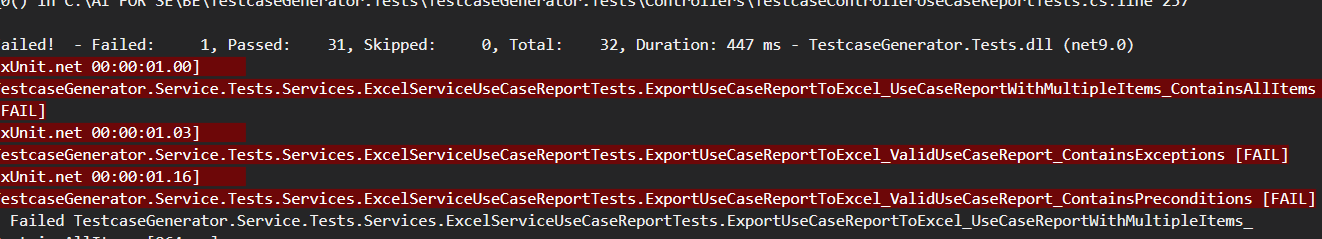
```

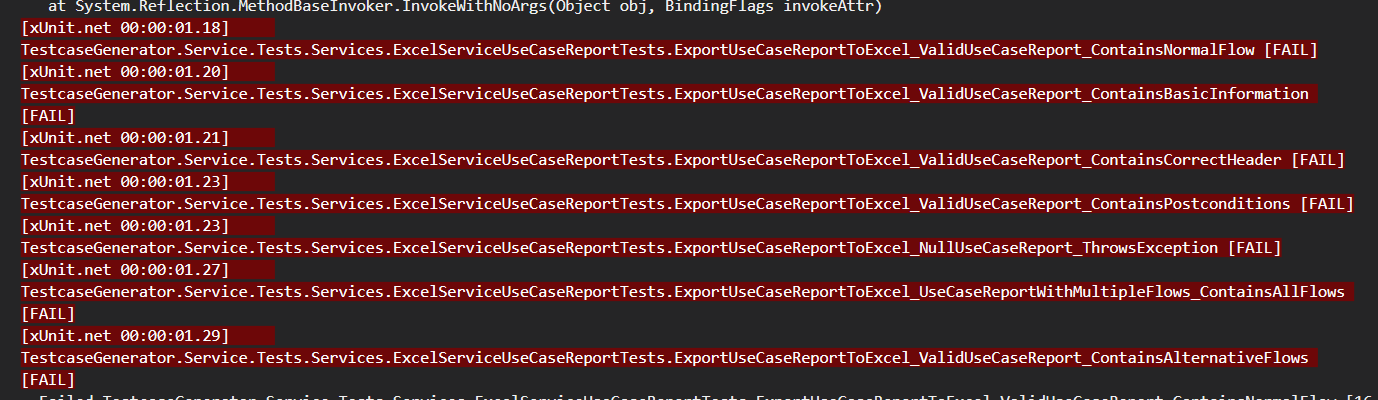
## Result:

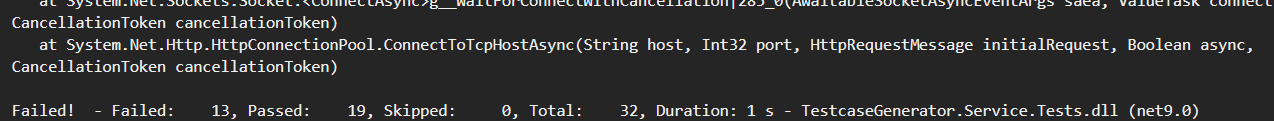
Test overall:

cmd: dotnet test









cmd: dotnet test --filter "Integration"

Passed! - Failed: 0, Passed: 12, Skipped: 0, Total: 12, Duration: 92 ms - TestcaseGenerator.Integration.Tests.dll (net9.0)

Test class:

cmd :dotnet test --filter "TestcaseControllerUseCaseReportTests"

Failed! - Failed: 1, Passed: 14, Skipped: 0, Total: 15, Duration: 612 ms - TestcaseGenerator.Tests.dll (net9.0)

cmd:dotnet test --filter "GeminiServiceUseCaseReportTests"

Failed! - Failed: 6, Passed: 11, Skipped: 0, Total: 17, Duration: 1 s - TestcaseGenerator.Service.Tests.dll (net9.0)

dotnet test --filter "ExcelServiceUseCaseReportTests"

Failed! - Failed: 10, Passed: 5, Skipped: 0, Total: 15, Duration: 1 s - TestcaseGenerator.Service.Tests.dll (net9.0)

## Error and Modify:

Prompt: (Input the error terminal), help me define error and code for each specific failed testcase and give me a modify way to update the test case to achieve better result

### 1. Missing Input Validation (Guard Clauses)

How to Fix:

You need to add "guard clauses" (input checks) at the beginning of your methods and constructors.

A) Fix GeminiService.cs Constructor: The constructor doesn't check for null dependencies, and it only checks for a null API key, not an empty one.

File: TestcaseGenerator\Service\[GeminiService.cs](http://geminiservice.cs)

// BEFORE ❌

public GeminiService(HttpClient httpClient, IConfiguration configuration)

{

\_httpClient = httpClient;

\_configuration = configuration;

\_apiKey = \_configuration["GeminiApi:ApiKey"] ?? throw new InvalidOperationException("Gemini API key not configured");

\_baseUrl = \_configuration["GeminiApi:BaseUrl"] ?? "...";

}

// AFTER ✅

public GeminiService(HttpClient httpClient, IConfiguration configuration)

{

\_httpClient = httpClient ?? throw new ArgumentNullException(nameof(httpClient));

\_configuration = configuration ?? throw new ArgumentNullException(nameof(configuration));

var apiKey = \_configuration["GeminiApi:ApiKey"];

if (string.IsNullOrEmpty(apiKey))

{

throw new InvalidOperationException("Gemini API key not configured");

}

\_apiKey = apiKey;

\_baseUrl = \_configuration["GeminiApi:BaseUrl"] ?? "...";

}

**B) Fix TestcaseController.cs Constructor:** Same as above, it's not checking for a null HttpClient.

*File: TestcaseGenerator\Controllers\TestcaseController.cs*

// BEFORE ❌

public TestcaseController(HttpClient httpClient, IConfiguration configuration)

{

\_geminiService = new GeminiService(httpClient, configuration);

}

// AFTER ✅

public TestcaseController(HttpClient httpClient, IConfiguration configuration)

{

// You should validate dependencies as they are passed in

ArgumentNullException.ThrowIfNull(httpClient);

ArgumentNullException.ThrowIfNull(configuration);

\_geminiService = new GeminiService(httpClient, configuration);

}

**C) Fix GeminiService.cs GenerateUseCaseReportAsync Method:** This method isn't checking its string inputs, which is why your tests are failing with a "No such host is known" error (it's trying to make a web request with a null value).

*File: TestcaseGenerator\Service\GeminiService.cs*

// BEFORE (at the start of GenerateUseCaseReportAsync) ❌

public async Task<UseCaseReport> GenerateUseCaseReportAsync(string useCaseName, string? additionalContext = null)

{

try

{

var prompt = CreateUseCaseReportPrompt(useCaseName, additionalContext);

// ... rest of the code

// AFTER (at the start of GenerateUseCaseReportAsync) ✅

public async Task<UseCaseReport> GenerateUseCaseReportAsync(string useCaseName, string? additionalContext = null)

{

if (string.IsNullOrWhiteSpace(useCaseName))

{

throw new ArgumentException("Use case name cannot be null or whitespace.", nameof(useCaseName));

}

try

{

var prompt = CreateUseCaseReportPrompt(useCaseName, additionalContext);

// ... rest of the code

**D) Fix ExcelService.cs ExportUseCaseReportToExcel Method:** This method is crashing with a NullReferenceException when report is null instead of throwing the expected ArgumentNullException.

*File: TestcaseGenerator\Service\ExcelService.cs*

// BEFORE (at the start of ExportUseCaseReportToExcel) ❌

public byte[] ExportUseCaseReportToExcel(UseCaseReport report)

{

// Add debugging to see what data we're working with

Console.WriteLine($"UseCaseReport Debug:");

Console.WriteLine($"UcId: {report.UcId}"); // This line crashes

// ...

// AFTER (at the start of ExportUseCaseReportToExcel) ✅

public byte[] ExportUseCaseReportToExcel(UseCaseReport report)

{

ArgumentNullException.ThrowIfNull(report); // Fail fast!

// Add debugging to see what data we're working with

Console.WriteLine($"UseCaseReport Debug:");

Console.WriteLine($"UcId: {report.UcId}");

// ...

### 2. Incorrect Excel Test Assertions (Type Mismatch)

How to Fix:

You need to explicitly convert the cell's value to a string in your assertions.

*File: TestcaseGenerator.Service.Tests\Services\ExcelServiceUseCaseReportTests.cs*

// BEFORE ❌

worksheet.Cell("A1").Value.Should().Be("II. Requirement Specifications");

worksheet.Cell("A3").Value.Should().Be("Created By:");

worksheet.Cell($"B{precondition1Row}").Value.Should().Be("PRE-1: User has valid account");

// AFTER ✅

worksheet.Cell("A1").Value.ToString().Should().Be("II. Requirement Specifications");

worksheet.Cell("A3").Value.ToString().Should().Be("Created By:");

worksheet.Cell($"B{precondition1Row}").Value.ToString().Should().Be("PRE-1: User has valid account");

// Or even safer with ClosedXML:

worksheet.Cell("A1").GetText().Should().Be("II. Requirement Specifications");

worksheet.Cell("A3").GetText().Should().Be("Created By:");

worksheet.Cell($"B{precondition1Row}").GetText().Should().Be("PRE-1: User has valid account");

### 3. Incorrect Test Data (Logical Error)

How to Fix:

Change the strings in the Should().Contain() assertions to match the data from CreateComplexUseCaseReport.

*File: TestcaseGenerator.Service.Tests\Services\ExcelServiceUseCaseReportTests.cs*

// In ExportUseCaseReportToExcel\_UseCaseReportWithMultipleItems\_ContainsAllItems

// BEFORE ❌

content.Should().Contain("PRE-1: User has valid account");

// AFTER ✅

content.Should().Contain("PRE-1: Customer has valid account");

content.Should().Contain("PRE-3: Customer has sufficient funds");

// In ExportUseCaseReportToExcel\_UseCaseReportWithMultipleFlows\_ContainsAllFlows

// BEFORE ❌

content.Should().Contain("1. User navigates to login page");

// AFTER ✅

content.Should().Contain("1. Customer selects payment method");

content.Should().Contain("5. System confirms payment success");