Integration Testing (.NET)

Agenda

1. INTRO
2. REPLACING THE EXTERNAL ARTIFACTS
3. COMPARISON
4. RESOURCES

1

Intro

What are Integration Tests?

Definitions

- Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network.

- Integration testing is done to validate the interaction between different modules.

- These tests ensure that component interactions work as expected against external artifacts like databases. Assertions can test component API, UI, or the side effects of actions like database I/O, logging, etc.

Introduction to Integration Tests

Integration tests evaluate an app's components on a broader level than unit tests. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:
- Database
- File system
- Network appliances
- Request-response pipeline

Integration Tests in ASP.NET Core

Integration tests in ASP.NET Core require the following:
- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual Arrange, Act, and Assert test steps:
1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The Arrange test step is executed: The test app prepares a request.
4. The Act test step is executed: The client submits the request and receives the response.
5. The Assert test step is executed: The actual response is validated as a pass or fail based on an expected response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

2

Replacing the External Artifacts

What Parts of the Software Can Be Replaced?

We don’t want these things to use the production environment, so we want to replace them:
- db
- file system
- queues
- third party services (APIs, email services, etc...)
- etc..

Database Replacing Strategies

- Worst: Different types of test doubles, mocking or stubbing DbContext and DbSet (repository pattern must be implemented)

- Better: an InMemory DB (but this isn't a complete copy from your DB, some features can't be tested or will behave differently)

- Best: using a real DB SQL/NoSQL (on the local machine, or Container-based technologies such as Docker) - creating the whole DB before running the tests

More about EF testing strategies
https://docs.microsoft.com/en-us/ef/core/testing/choosing-a-testing-strategy

Database Replacing Comparison

Feature	In-memory	SQLite in-memory	Mock DbContext	Repository pattern	Testing against the database
Test double type	Fake	Fake	Fake	Mock/stub	Real, no double
Raw SQL?	No	Depends	No	Yes	Yes
Transactions?	No (ignored)	Yes	Yes	Yes	Yes
Provider-specific translations?	No	No	No	Yes	Yes
Exact query behavior?	Depends	Depends	Depends	Yes	Yes
Can use LINQ anywhere in the application?	Yes	Yes	Yes	No*	Yes

* All testable database LINQ queries must be encapsulated in IEnumerable-returning repository methods, in order to be stubbed/mocked.

3

Comparison

Manual vs Integration Tests

The bad scenario is that the API is tested by triggering the request via the actual application. This can be time-consuming in multiple ways:
- It's required that the whole environment (or most of it) is up
- It's sometimes not easy to trigger the request, for example, stepping through a process, or to fill out a form

With an integration test, a test needs to be written once and only needs to be manually checked once. It has the added benefit that:
- It's available for the whole team
- Can be tested (and written) separately, apart from the application
- It's fast to run
- No extra tooling needs to be used

Unit vs Integration Tests Setup

Writing unit tests usually means mocking or stubbing an interface in the application. For more complex scenarios, this setup can be long and will become unreadable over time.
While in the integration tests, we're doing the setup once per all tests (setting up the infrastructure, back door).

But the integration test needs to wait for the whole workflow to be implemented.
While the unit test can be done for each small part independently.

Integration vs E2E (End to End) Tests

E2E tests everything (not just the code integration like integration tests), it uses sandbox database, third party services (APIs, emails) which are always up (they aren't created only when the test runs).

E2E test requires the test to setup the system components as they are in production. Real database, services, queues, etc. The reason for this is to see that your system is wired correctly (database connections, configuration and such).

6

Resources

Microsoft Docs

Integration tests in ASP.NET Core
https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-6.0

Entity Framework

Testing EF Core Applications
https://docs.microsoft.com/en-us/ef/core/testing/

Choosing a testing strategy
https://docs.microsoft.com/en-us/ef/core/testing/choosing-a-testing-strategy

Testing against your production database system
https://docs.microsoft.com/en-us/ef/core/testing/testing-with-the-database

Testing without your production database system
https://docs.microsoft.com/en-us/ef/core/testing/testing-without-the-database


endava