Using Integration Tests to Validate Persistence

Julie Lerman

EF Core Expert and Serial DDD Advocate

@julielerman | thedatafarm.com

Overview

Test how EF Core is performing persistence operations

Store, Update, Delete and Retrieve

Test areas of the aggregate

Integration tests!

Tweak the mappings and aggregate as needed

Testing with the SQL Server Provider vs. the In-Memory Provider

These Tests Are for Exploring the DB Interaction

I'll be using the SQL Server provider,
not the In-Memory provider.

Why Integration Test with SQL Server, Not In-Memory

- In-memory is only a set of lists and limited in functionality
- Many RDBMS features are not supported by in-memory
- See the true queries, expose explicit issues with actual target DB
- In-memory has its place, just not for this specific purpose.
- There have even been calls from the community to remove the I-M provider!

Remove the in-memory provider #18457

New issue

Closed ajcvickers opened this issue on Oct 18, 2019 · 40 comments

ajcvickers commented on Oct 18, 2019 Member

~~Note: the decision to do or not do this has not been made~~. Feedback is appreciated.

While the in-memory provider is marketed as "for testing" I think that is somewhat misleading. Its actual main value is as a simple back-end for all our internal non-provider-specific tests.

When we started EF Core, it seemed like it would also be useful for testing applications. This can work with appropriately factored test code and a good understanding of where the provider behavior is different. However, I haven't recommended this for many years, and I don't think anyone else on the team would recommend it either. The main reason is that it's too easy to get caught out by differences in provider behavior, and therefore not realize that the behavior of your test is different from the behavior in production. This can be mitigated by using a provider that is closer to what is being used in production--for example, SQLite. However, the only way to know if something really works is to test with the same code that is running in production.

So, if the only real, non-pit-of-failure approach is to not use the in-memory provider for testing, then maybe we shouldn't ship in 5.0?

Assignees

ajcvickers

Labels

area-in-memory breaking-change closed-out-of-scope type-enhancement

Projects

None yet

Milestone

No milestone

Setting Up the Tests

```
{create some objects and add to the context instance}

_context.SaveChanges();

_context.ChangeTracker.Clear();

var theContractsFromTheDatabase=_context.Contracts.ToList();
```

These Tests are Designed to Validate EF Core's Persistence

ChangeTracker.Clear() is very convenient for testing...
...but not recommended for production code

Ensuring Basic Data is Getting Stored Correctly

EF Core 7: Use Raw SQL to Query Scalar Data

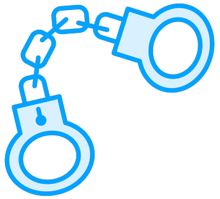
```
var value = _context.Database
        .SqlQuery<bool>($"SELECT [_hasRevisedSpecSet] FROM [ContractVersions]")
        .ToList();
```

```
var value = _context.Database
       .SqlQuery<bool>($"SELECT [_hasRevisedSpecSet] FROM [ContractVersions]")
       .ToList();
```

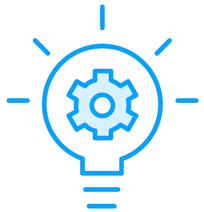
If composing other LINQ operators, you need to return the column as Value

```
var value = _context.Database
       .SqlQuery<bool>($"SELECT [_hasRevisedSpecSet] as [Value] FROM [ContractVersions]")
       .FirstOrDefault();
```

EF Core 7: Use Raw SQL to Query Scalar Data



Until EF Core 7, you can only query entities with raw SQL



DbContext.Database.SqlQuery<T>(formattable string). ToList()

You can compose other LINQ methods, but the scalar has to be named “Value”

```
_context.Database.SqlQuery<T>
 ($“SQL Expression").ToList();
```

- Basic syntax for EF Core 7 generic SqlQuery<T> with a formattable string

```
_context.Database.SqlQuery<bool>
  ($"SELECT [_hasRevisedSpecSet]
     FROM ContractVersions")
  .ToList();
```

- Retrieving all of the _hasRevisedSpecSet values

```
_context.Database.SqlQuery<bool>
  ($"SELECT [_hasRevisedSpecSet] as
    [Value] FROM ContractVersions")
 .FirstOrDefault(v=>v.ContractId==1);
```

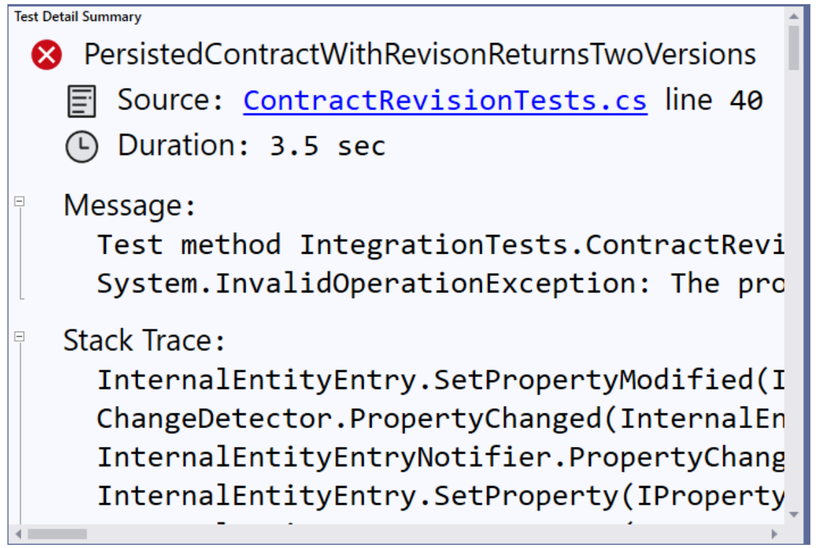
- Composing with FirstOrDefault()
- Notice _hasRevisedSpecSet is being named “Value”

Using JSON to Test Persistence of a Full Aggregate

Default Testing Contract Aggregate

Preparing to Test Contract Revisions

Interpreting the EF Core's Weird Exception Message

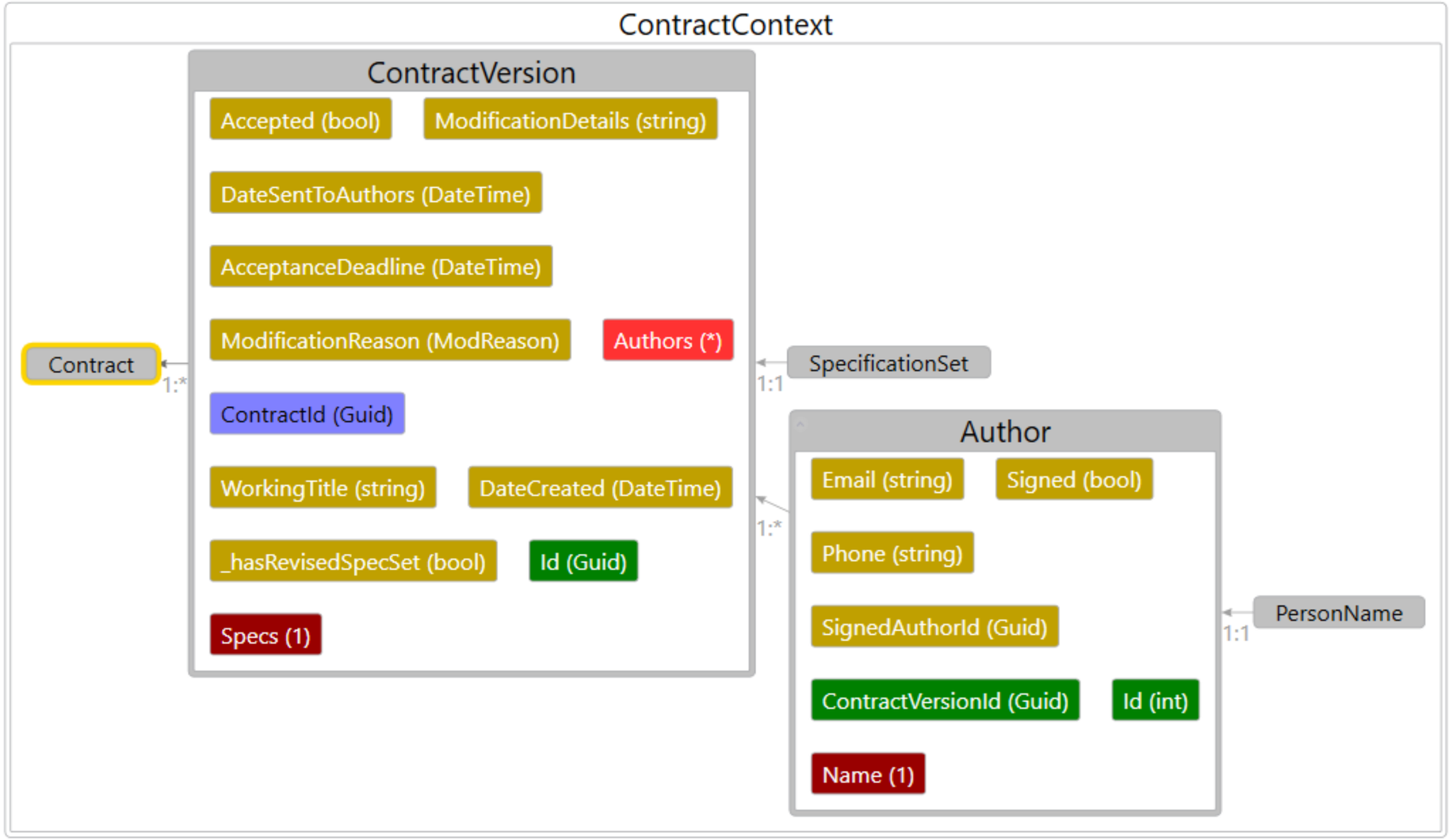


The property 'Author.ContractVersionId' is part of a key and so cannot be modified or marked as modified.

To change the principal of an existing entity with an identifying foreign key, first delete the dependent and invoke 'SaveChanges', and then associate the dependent with the new principal.

“"Author.ContractVersionId' is part of a key “

and so cannot be modified or marked as modified.



Responding to SaveChanges()

DetectChanges()

1

Determine SQL

2

Send command to DB

3

“Sometimes you just have to google it...”

- Julie Lerman

The logic isn't bad, but it is tripping up EF Core's trickery in persisting owned entities!

Communication Problem

“Copy”

...was missing from the ubiquitous language!

Refactoring the Domain and Adjusting the Mappings

Be Wary of Domain Changes to Benefit Persistence

You don’t want to change your domain logic for the sake of persistence

But if persistence happens to lead you to domain improvements, that’s a bonus!

Add Copy Methods to the Value Objects

Author.Copy()

SpecificationSet.Copy()

Refactor logic to use those new methods

```
static class Extensions
{
    public static IList<T> Clone<T>(this IList<T> listToClone) where T: ICloneable
    {
        return listToClone.Select(item => (T)item.Clone()).ToList();
    }
}
```

Usage

```
v1.Authors.Clone();
v1.Specs.Clone();
```

Alternative: Create an Extension Using .NET's ICloneable Interface

Thanks to Pluralsight author, Roland Guijt, for the idea and even the sample code!

New Error is a SqlException

The property 'Author.ContractVersionId'is part of a key and so cannot be modified or marked as modified.

The INSERT statement conflicted with the FOREIGN KEY constraint "FK_ContractVersion_Authors_ContractVersions_ContractVersionId". The conflict occurred in database "PubContractTests", table "dbo.ContractVersions", column 'Id'.

First problem resolved with Copy()

EF Core is happy now,
but SQL Server isn't!

My EF Core Debugging Workflow

Check SQL

Check EF Core logging messages

Step through the code in debug and look for anomalies

Explore debug info while stepping through code

The cause:

An oversight in the aggregate logic

We didn’t set the ContractId in ContractVersion’s constructor!

And … no unit test to uncover it.

The Same SqlException, But a Yet Another Issue

The property 'Author.ContractVersionId' is part of a key and so cannot be modified or marked as modified.

First problem resolved with Copy()

The INSERT statement conflicted with the FOREIGN KEY constraint "FK_ContractVersion_Authors_ContractVersions_ContractVersionId". The conflict occurred in database "PubContractTests", table "dbo.ContractVersions", column 'Id'.

Because I wasn't setting ContractId in ContractVersion!

The INSERT statement conflicted with the FOREIGN KEY constraint "FK_ContractVersion_Authors_ContractVersions_ContractVersionId". The conflict occurred in database "PubContractTests", table "dbo.ContractVersions", column 'Id'.

Version is being marked "Modified" for a different reason

EF Core is Following Its Own Rule

Detect changes honors store-generated keys (default) over explicitly set values

ContractVersion is a dependent in a graph of tracked entities

Configure the property as ValueGeneratedNever

We’re fixing the EF Core problem in EF Core, not in our domain logic

```
[TestMethod]
public void PersistedContractWithRevisonReturnsTwoVersions()
{
    CreateNewContractAndAddRevision();
    _context.SaveChanges();
    _context.ChangeTracker.Clear();
    var contractFromDB = _context.Contracts.Include(c => c.Versions)
        .FirstOrDefault();
    Assert.AreEqual(2, contractFromDB.Versions.Count());
}
```

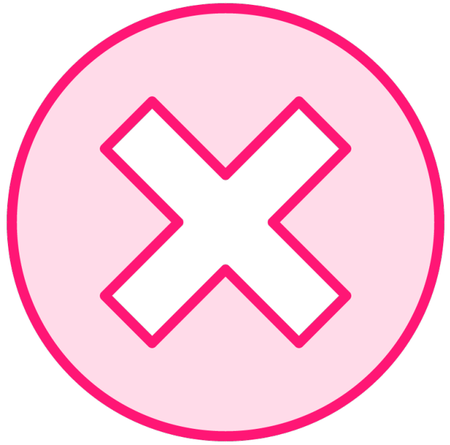
Integration Testing Against Target DB is Critical when Mapping EF Core

This one little test, counting objects, revealed so many important issues.

- Mapping issues
- Domain logic issues
- A bug in the domain logic (which could have been caught with a unit test!)

Exploring Some More Revision Integration Tests

Automated Testing Practice



Create failing tests...

...to have more confidence in them when they pass!

Summary

Integration testing against the target database is critical.

EF Core configuration can solve most persistence problems.

EF Core has some quirks around owned entities, aka our value objects.

JSON is a great way to assert that deep graphs match.

Take advantage of logs and debug info to solve exceptions.

Discovered issues with domain code

Resources

Mapping code generated keys: https://bit.ly/EFCore_GenKeys

Querying Scalars in EF Core 7: https://bit.ly/EFCore_QueryScalar

How to write custom converters for JSON serialization in .NET
https://bit.ly/netcore_jsonconvert

GitHub Issue re removing the In-Memory provider
https://github.com/dotnet/efcore/issues/18457

Up Next:

Reasoning About Many-to-Many Variations