# Cworks.DBTest

A .net core option for exercising DB Objects. Perform isolated, repeatable unit and contract tests against your SQL Server DB objects.

## Background

Testing is important. It ensures expected behavior when adding new functionality and regression tests ensure we don’t break existing functionality. Good tests exercise both the expected “Happy path” and alternate execution paths.

Testing should be exercised often. When a change is introduced, we want to know as soon as possible if that change has changed contract or broken behavior. Full test execution often happens as part of a continuous integration (CI) pipeline. Some IDEs, like Visual Studio, offer an option to perform live testing, constantly running tests in the background when changes are detected.

Tests should occur at ever layer in the application. We test have unit tests for the controller, business service and data layers. We have Pact testing to ensure the contract hasn’t broken, and integration testing ensures the layers and services work together to perform as expected.

Traditionally tests are focused on our code and UI, but what about the database? Typically, there is no mechanism for testing DB objects used in our services. Specifically, how do we test our database tables, views, stored procedures, and functions? For some services, business logic is performed at the database layer, and yet we have few options to test their functionality and contract.

### The challenges of writing Db Tests

Without automated tests, introducing change in the database layer can introduce risk and unseen fragility into the service offering. Writing DB tests for services utilizing schema migrations can be challenging because though migrations happen after the code unit tests have been run. In some cases, migrations are run as part of the application startup, and without a custom test pipeline, there is no opportunity to deploy migrations without deploying the code.

The structure of the Db does not lend itself to writing good unit tests. Good unit tests have the following characteristics:

|  |  |
| --- | --- |
| **Characteristic** | **Challenge for Db Tests** |
| Arrange, Act, Assert | Lack of tooling to support this best practice |
| No Interdependent Tests | The shared state of data in a database could lead to one test relying on the output of another |
| Tests are repeatable. | Any tests that do writes to the database could impact their ability to be repeated |
| Tests are deterministic | Shared state allows tests to alter the results of their own and other tests |
| Fast execution | The inability to mock data stores requires real I/O and leads to longer execution times |

## Introducing DbTests

DbTests is .NET Core testing framework designed to support testing of DB objects

* Your Db Tests are implemented as a series of .NET classes; one test per class.
* Can leverage both the EF Data Context and ADO.NET commands interchangeably
* Enforces best practices with an Arrange, Act, and Assert pattern
* Ensures state isolation. Allowing tests to be fully independent, repeatable, and deterministic.
* As an xUnit test, your DbTests can run as part of continuous integration and local development.
* Enables automated regression testing for Db Objects.
* Test out your EF Data migrations before they apply to the existing databases.
* Has zero impact on the existing databases (no leakage).

## How it Works

DbTests leverage XUnit as a testing framework. This means you can author dbtests the same way you author code tests, with .net code.

As part of test execution, a DbTest fixture is initialized. This creates an empty database, runs the existing migrations to bring the database up to the latest schema, including seed data and then runs the authored db tests.

Tests are run inside a transaction scope. This transaction scope provides test isolation. No data state changes from any test can impact any other. At the end of each test, the state of the database is reverted back to its migration and seeded state. No remnants from the executed tests remain.

When the tests are completed, the test database is torn down, test results are gathered and reported via the XUnit test runner.



## Getting Started

### Add the package reference to your test project.

Cworks.DbTests supports tests MS Sql Server installations and comes in two flavors; Entity Framework and ADO.Net.

If the application is a .NET Core application that uses Entity Frameworks, add a reference to **Cworks.DbTest.EFCore.**

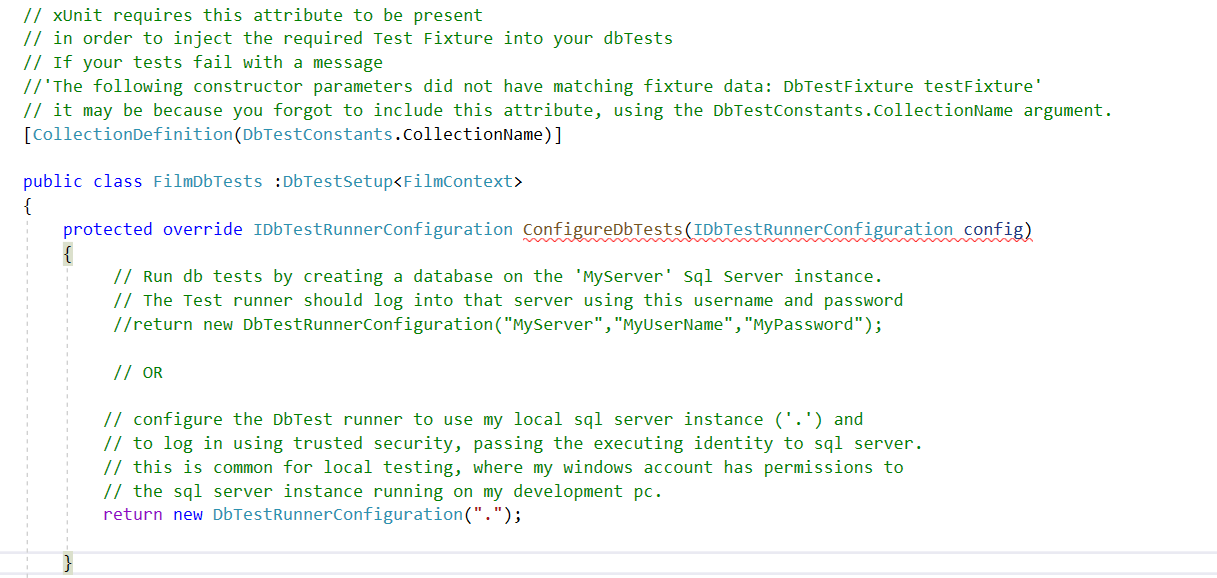
If the application uses ADO to access its database, add a reference to **Cworks.DbTests.ADO**

### Configure Your Test Environment

When DbTests are run a new database is created on the server of your choice. After the tests are run, this test database is removed. The DbTestSetup class allow you to specify the connection information to the DB server that should be used when running your dbTests.

Create a class that extends DbTestSetup<YourDataContext>. Override the ProduceDbTestConfiguration method and supply the login credentials to the server to use for testing. Note: The credentials specified must have permissions to create and drop a database on this server.

For example, to configure db tests for our FilmDb data context, we might author the following:



Let’s review this code.

1. The code has been decorated with a the **[CollectionDefinition(DbTestConstants.CollectionName)]** attribute. This is required, as it instructs the XUnit test runner to use this class when running your DbTests.

If you forget to include this attribute you’ll see an error when attempting to run the tests 'The following constructor parameters did not have matching fixture data: DbTestFixture testFixture'

1. The class extends the DbTestSetup base class and specifies the data context you’ll be testing, FilmContext.
2. The overridden ConfigureDbTests method returns a new DbConfiguration instance. This instance holds all the data about how the tests runs are configured, including login credentials. The constructors support specifying the server name and either user credentials, or trusted security,

We will revisit this code later as we customize some behavior of the test runner.

## Author Your Tests

Writing a Db test begins with deciding the type of the DB Object under test and the objective of the test. DbTests offers a variety a different base classis to speed test development.

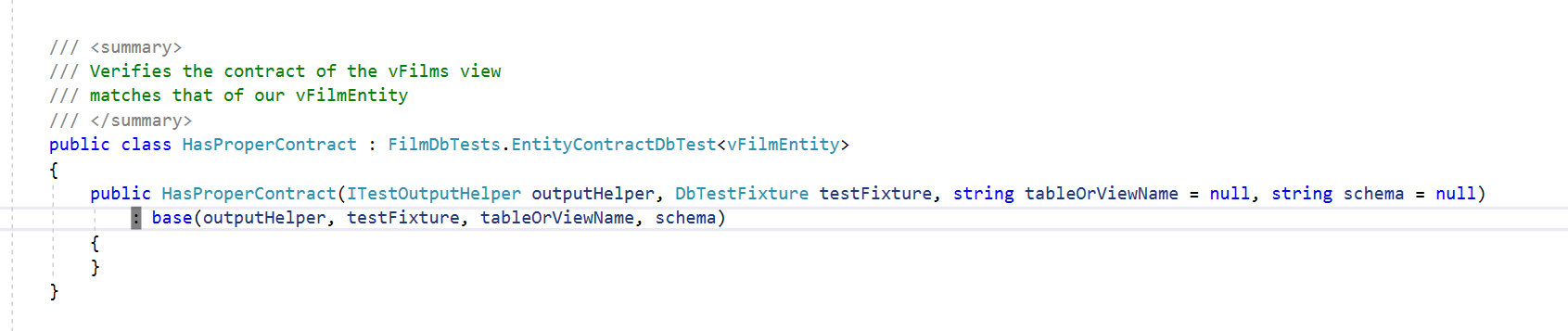
|  |  |  |
| --- | --- | --- |
| **To Verify the** | **Of Your** | **Write a test class that extends** |
| Contract | View | EntityContractDbTest<Tentity> |
| Contract | Stored Procedure | StoredProcedureContractDbTest |
| Contract | Table-Value Function | TableFunctionContractDbTest |
| Behavior | View, Stored Procedure, Scalar Function, Table-Value Function | DbTest |

#### Verifying View definitions

It can be useful to ensure the schema definition of your view matches the expected layout. If a migration changes the definition of the view, by adding, dropping or renaming columns, contract verification tests will fail, alerting you to the problem before the code is ever deployed.

Writing a test to verify the contract of a view definition is accomplished by creating a test class that extends from the EntityContractDbTest for your view entity.

For example, the following test class will ensure the actual contract of the view underlying the vFilmEntity view matches the class definition of the vFilmEntity class.



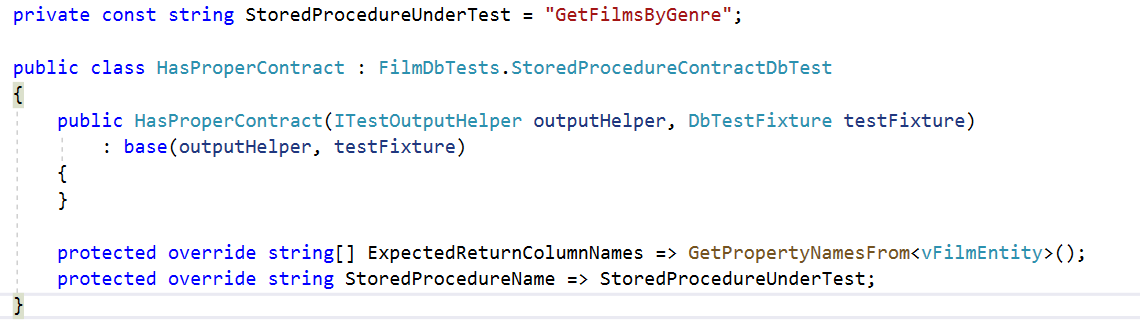
The class extends the EntityContractTest of your configured DbTest class, and accepts a generic of the entity to verify. The above example will verify the contract of the underlying SQL View for the vFilmEntity exactly matches the public definition of the vFilmEntity class. The test extends the EntityContractDbTest of our configured DbTest class, FilmDbTests.

By default, the test will use the view defined in the [Table] attribute of the entity class for view. This can be overridden by specifying the optional table or dbname and schema in the constructor.

#### Verifying Stored Procedure contracts

To ensure the contract of the data returned from your stored procedure matches the contact of your entity class, write a test that extends the StoredProcedureContractDbTest class.

In our sample application, Film entities can be populated from the GetFilmsByGenre stored procedure. We can write a test to ensure that stored procedure returns the columns expected by our FilmEntity class as follows:



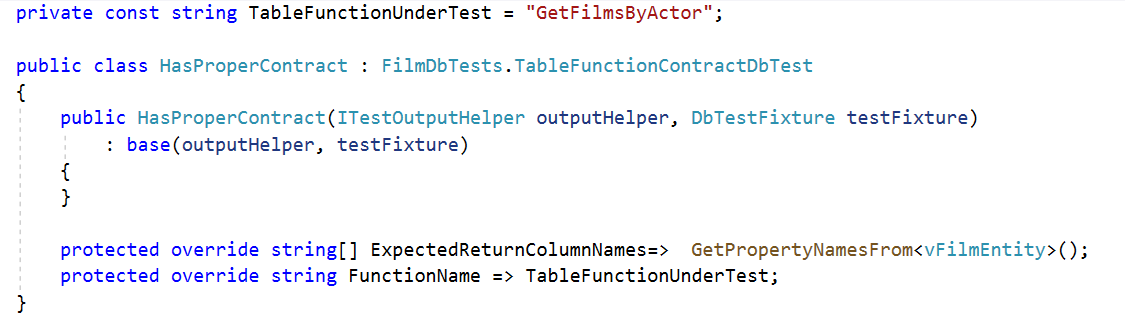
The test class extends the StoredProcedureContractDbTest in our configured TestDb class, FilmDbTests. The overrides for ExpectedReturnColumnNames are populated directly from the vFilmEntity class via our GetPropertyNamesFrom<> helper, and the override for StoredProcedureName specifies the name of the stored procedure we’re testing.

If the stored procedure is expected to return a different set of columns than those defined in an entity, the list of expected return column names can be customized in the method override.

#### Verifying Table-Value function contracts

To ensure the contract of the data returned from your table-value function matches the contact of your entity class, write a test that extends the TableFunctionContractDbTest class.

In our sample application, Film entities can be populated from the GetFilmsByActor table-value function. We can write a test to ensure that function returns the columns expected by our FilmEntity class as follows:



The test class extends the TableFunctionContractDbTest in our configured TestDb class, FilmDbTests. The overrides for ExpectedReturnColumnNames are populated directly from the vFilmEntity class via our GetPropertyNamesFrom<> helper, and the override for FunctionName specifies the name of the table value function we’re testing.

If the function is expected to return a different set of columns than those defined in an entity, the list of expected return column names can be customized in the method override.

#### Verifying the behavior of the db objects

In addition to ensuring the contract of the db object matches the expectations, we need to verify the behavior of the objects matches the expectations.

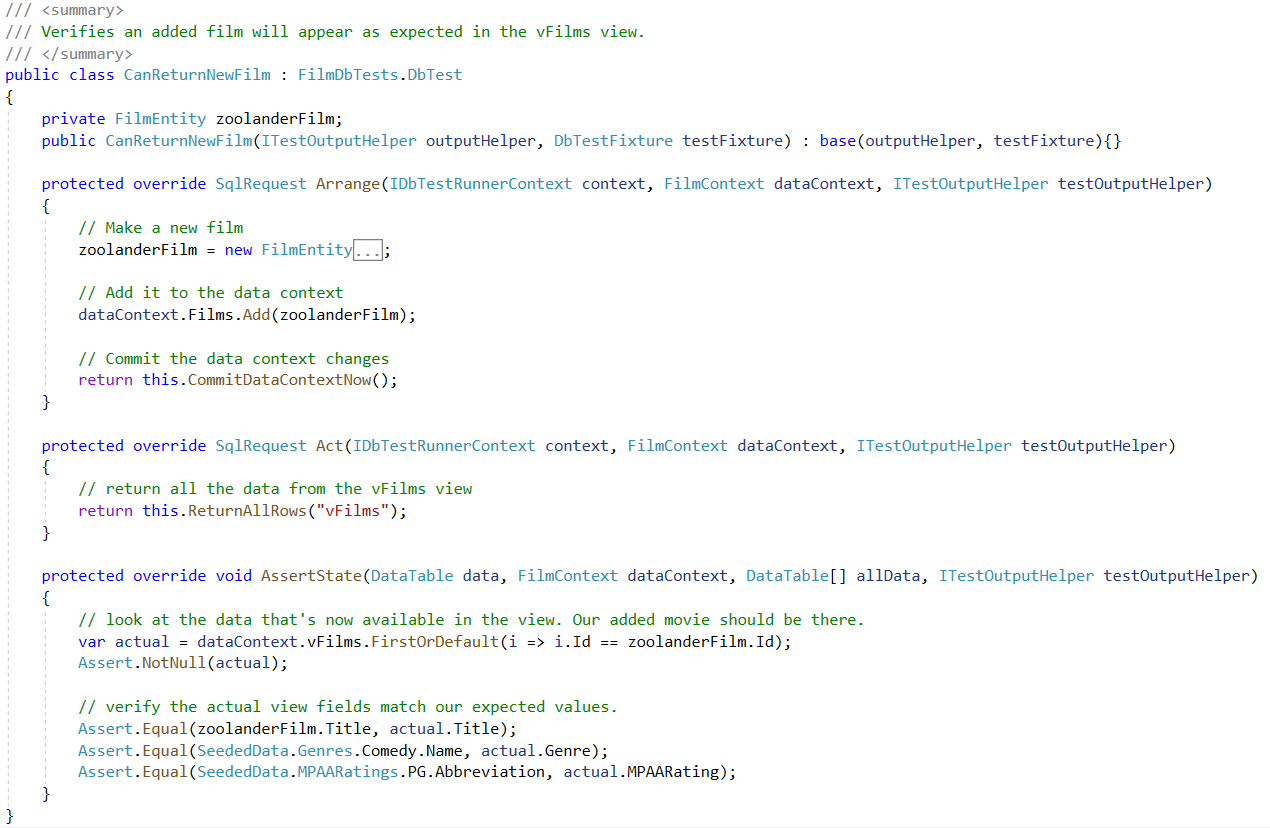
The best way to verify the behavior of an object is to write a test that configures the item under tests, excercises it, and then validates the output. The best practice pattern for this is Arrange/Act/Assert.

You can author tests to verify the functionality of a db object using the Arrange/Act/Assert pattern implemented in the DbTest class.

Returning to our example, let’s assume we want to verify that when we add a film to the tables in the database, the film appears in our film view.

We begin by creating a class that extends from DbTest in our configured DbTest class, FilmDbTests.

The class overloads provide methods for our Arrange, Act and Assert actions.



In the arrange method, we add a new film to our database. Since our application is using Entity Framework, we can easily add a new film by creating a new FilmEntity instance for the film to be added, add it to the data context and return a sql request response which tells the dbtest framework to commit our data context now.

After arranging our data, the Act method is invoked, and here we tell the test framework that we want to return all the rows from our vFilms view.

The arrange method is where we confirm our expectations. We confirm them using standard XUnit assert statements. Note the arguments of the method include our data context, so its easy to find the new film in our vFilm view by searching for it in the vFilms dbset on the data context.

We assert that we found the film and that the films properties match what we expected.

#### Working with SqlRequest

Note the return type of the Arrange and Act methods are of type SqlRequest. SqlRequest tells the DB

Test framework what additional operations should be performed after running your method code.

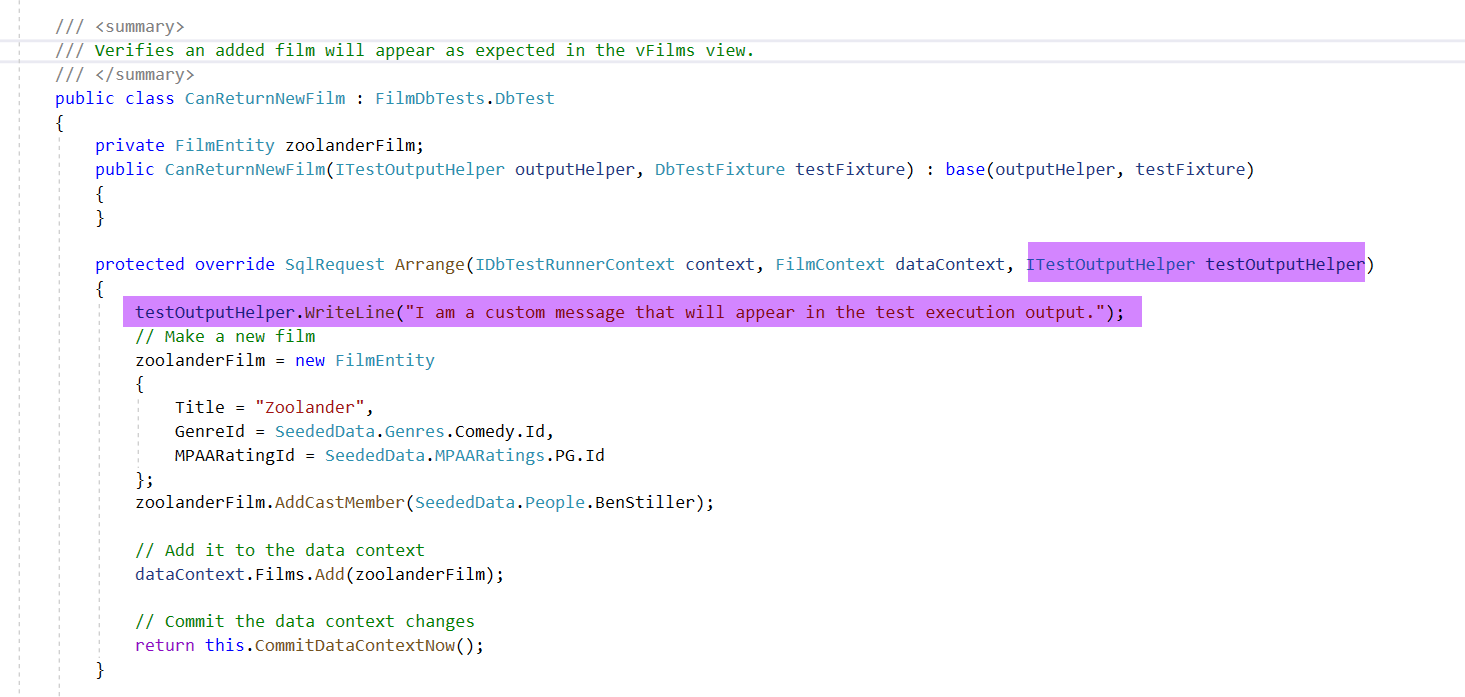
|  |  |
| --- | --- |
| **SqlRequest Helper** | **Description** |
| ReturnAllRows | returns all the data from the specified table or view, passing the returned rows to the Assert method. |
| InvokeScalarFunction | Executes the specified function and passes the output values to the Assert method. |
| InvokeTableFunction | Executes the specified table function and passes the returned rows to the Assert method. |
| ExecuteStoredProcedure | Executes the specified stored procedure and passes the procedure output data to the Assert method. |
| CommitDataContextNow | Commits any data added to the data context to the database |
| RunSqlText | Executes the specified Sql commands and passes the output to the Assert method |
| ReturnNoRows | Executes the specified command, but passes no returned data to the Assert method. |
| ReturnRowCount | Executes the specified command, and returns the row count to the Assert method. |
| DoNothing | Performs no additional actions. |

DbTests offer a variety of SqlRequest responses including:

## How do I?

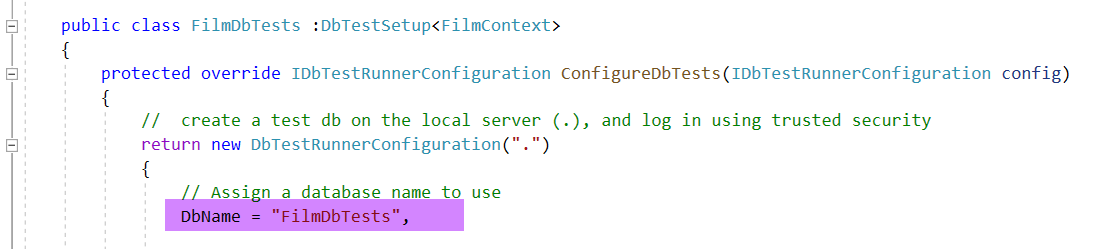
### How do I add custom messages to the test output?

Db Tests logging and exceptions are always logged and visible in the output of the test run. Developers can augment the logged data by invoking the WriteLine() method on the provided TestOutputHelper to include additional data in the test output.



### How do I specify the name of the DB used for testing?

The DbTest runner must be configured with the name and login credentials to the SQL Server instance to use for testing. By default, it will create a unique Database name for each run. You can override this behavior by specifying the DbName property in the DbTestSetup class:



This instructs the DbTest runner to use a database name called “FilmDbTests” instead of a dynamic name. If the already database exists, the DbTest runner will use the existing database and will attempt to run migrations against it, bringing it up to the expected schema definition and seed data.

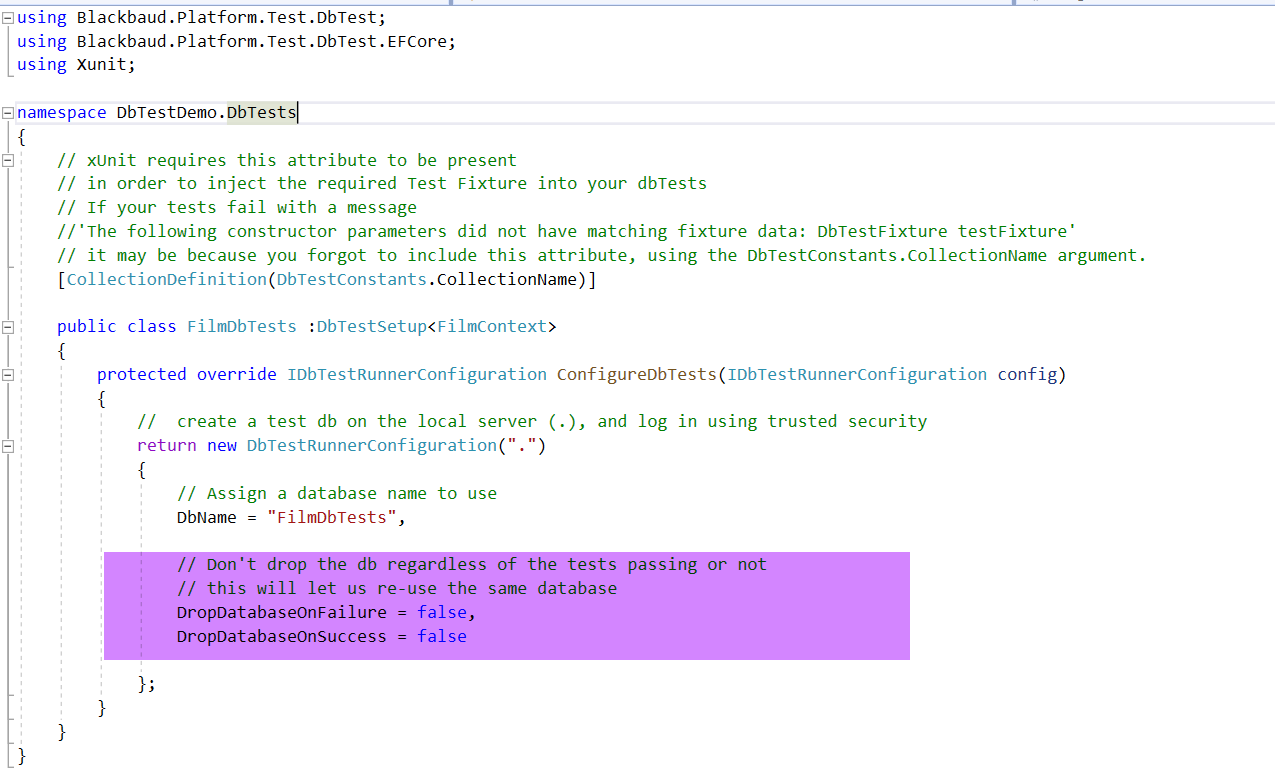
### How do I keep my test DB around between runs?

By default, Db Test runner will create and migrate a new database every time the tests are run. When the tests are complete, the test runner drops the database from the test DB server.

You can configure the Db Test setup class to retain this database between runs. By retaining the database between runs, the Db Test runner won’t need to provision a new database or run any additional migrations the next time the tests are run, speeding the tests execution.

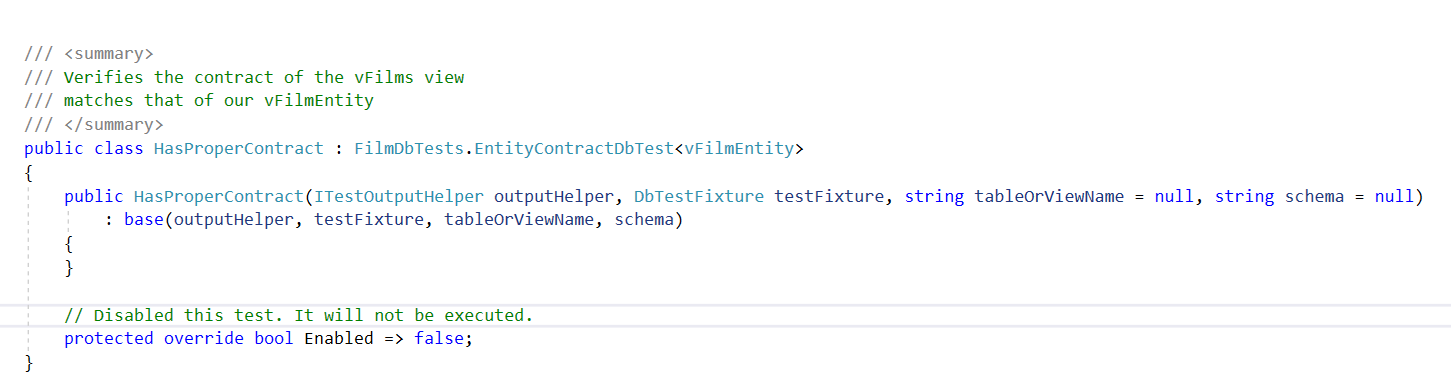
You may configure the DbTest runner to keep the database 1) if any of the tests fail, or 2) When all the tests complete 3) Regardless of the test passing or not.

To configure the DbTest runner to keep the database, modify the DbTestRunnerConfiguration as follows:



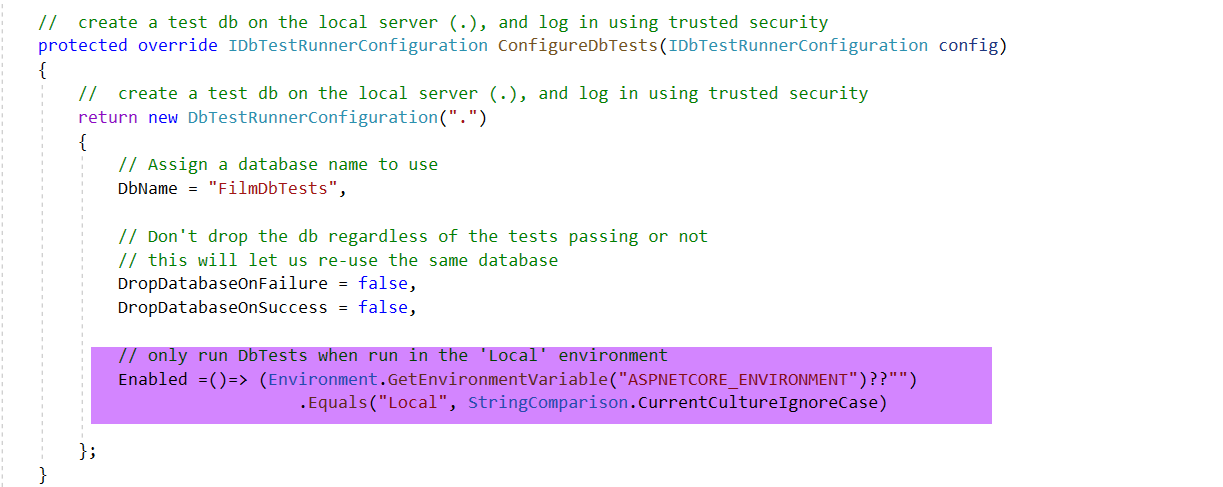
### How do I control which tests are executed, and when?

By default, all tests are enabled. You can disable a single test by overriding the Enabled property and setting its value to false, like so:



You can also configure the DbTest runner when it should run any of its tests. This could be useful if, for example, you want to run your tests locally, but do not run them as part of the CI pipeline builds.

The configure the ability to run any tests, override the Enabled lambda in the DbTestSetup class and supply your logic:



## Tips, Tricks and Caveats

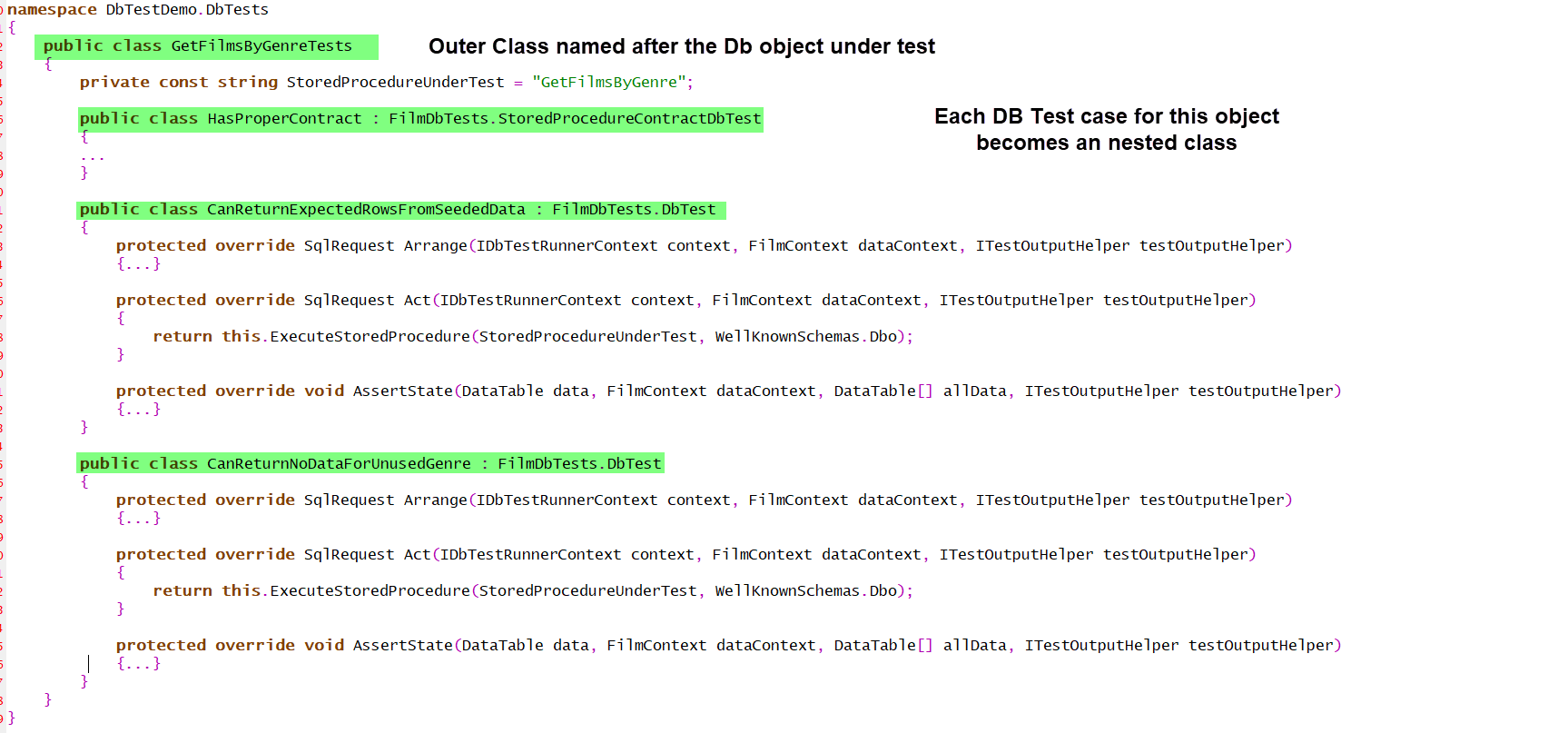
### Tip: Speed up local testing

You can configure the DbTest runner to use a specified DBName and to keep that database between test runs. This will speed test execution because the system won’t need to create and provision the database for every dbtest run and then tear down that database at the end of the test run.

Speeding up the test cycle can be helpful in local development, especially if Live Testing is enabled.



### Tip: One (outer) class to rule them all

Create a public class for each DB object you want to test. Nest all your test classes related to that object inside the public class. When tests run, it will be easy to know exactly which objects were being tested and the test runner will do a good job organizing the test run results. 

### Tip: Store the DbObject name as a constant

Create a private constant to hold the name of the object under test. This makes for cleaner tests, easier re-use and let maintenance if the name of the object changes.



### Trick: Troubleshooting failed tests

If a db test fails, it can be challenging to troubleshoot because the impact of the test is removed at the end of every test run.

You can configure a DbTest to retain its data changes after the test is completed. Doing so allows the develop to access the database outside the test run and troubleshoot its behavior.

Please note – if you retain the changes to the data after the test run, you should manually drop the test db before the next run, as your retained changes have polluted the state of the data and may cause unpredictable results.

To configure the DbTest to retain its data after the test run, set the CommitTransactionScope property of the DbTestRunnerContext to true.



### Caveat: Code under test should respect transaction boundaries.

Because the DbTests run code inside a transaction scope, using override hints, like ReadUncommitted may result in unpredictable results.

### Caveat: Configure the DbTest setup class with proper credentials.

The credentials specified in the DbTest setup class must have permissions to Create a database, Create objects within the database, read, write, and execute objects within that database, and finally it must be able to drop a database on the specified server.

### Caveat: Single DbTestClass support.

Currently the DbTest runner only supports running for a single data context per solution. This may change in the future if there is demand for supporting multiple data contexts.