Masaryk University
Faculty of Informatics

Working with Complex SQL Queries in Unit Tests

Bachelor's Thesis

Dávid Urbančok

Brno, Fall 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Dávid Urbančok

Advisor: Bruno Rossi, PhD

Acknowledgement

I would like to thank my supervisor, Bruno Rossi, PhD, who has helped me a lot with various invaluable inputs into this thesis. I would also like to extend my gratitude to Mgr. Michal Pietrik for countless hours of consultation and help with the implementation.

Abstract

This thesis is aimed at analyzing and extending the existing support of faking objects in Kentico's automated unit tests and its insufficiency when performing in-memory join operations on these objects.

After the analysis, a prototype solution that allows the joining to be performed in the memory is proposed and implemented.

This solution will allow rewriting some integration tests to unit tests in order to eliminate the need to run these tests against a database. This will increase performance and decrease the running time of the automated tests in Kentico's continuous integration tool.

Lastly, performance measurements are made to determine the time gain of this prototype and an estimation is calculated how much computing time is saved.

Keywords

Kentico, unit testing, integration testing, Info object, InfoProvider object, DataSource, NUnit framework, performance

Contents

1 Introduction **1**

2 Kentico's CMS structure **3**
2.1 *Kentico back-end* . . . 3
2.1.1 Info class . . . 3
2.1.2 InfoProvider class . . . 4

3 Kentico DataQuery API **5**
3.1 *DataQuery* . . . 5
3.2 *ObjectQuery* . . . 5
3.3 *DataQuery concept* . . . 5
3.3.1 DataSource property . . . 8
3.3.2 DataSet . . . 8
3.4 *DataQuery Join types* . . . 9
3.5 *Current implementation* . . . 10
3.5.1 The Source method . . . 10
3.6 *DataQuery insufficiencies* . . . 11

4 Test automation **13**
4.1 *Test automation benefits* . . . 13
4.2 *Unit tests* . . . 13
4.3 *Integration tests* . . . 14
4.4 *Mimicking real objects in automated tests* . . . 15
4.4.1 Mocks vs. Fakes vs. Stubs . . . 16

5 Automated testing in Kentico **17**
5.1 *Types of tests in Kentico* . . . 17
5.1.1 Unit tests . . . 17
5.1.2 Integration tests . . . 17
5.1.3 Isolated integration tests . . . 17
5.2 *Faking Info and InfoProvider objects in unit tests* . . . 18
5.3 *NUnit framework* . . . 19
5.3.1 Nunit attributes and life cycle . . . 19
5.3.2 Assertions . . . 20
5.3.3 Constraint-based assert Model . . . 21

6 Solution and performance measurements **22**
6.1 *Solution* . . . 22
6.1.1 Faking the class structure . . . 22
6.1.2 Implementing logic to the `Source` method . . . 23
6.1.3 Implementing a new layer of abstraction . . . 23
6.1.4 Implementing the logic of joining . . . 24
6.1.5 Setting the `DataSource` property . . . 25
6.2 *Performance measurements* . . . 26
6.2.1 Running times of 100 automated tests . . . 27
6.2.2 Running times of 500 automated tests . . . 28
6.2.3 Running times of 1000 automated tests . . . 29
6.2.4 Conclusion . . . 30
6.2.5 Testing hidden joins . . . 30
6.3 *Extensions and alternative solutions* . . . 31
6.3.1 Extensions . . . 31
6.3.2 Alternative solutions . . . 32

7 Conclusion **33**

Bibliography **34**

A Implementations **36**
A.1 *MemorySource method* . . . 36
A.2 *MemorySource class* . . . 37
A.3 *PrepareDataSources* . . . 38
A.4 *Inner join* . . . 39
A.5 *Left join* . . . 40
A.6 *CombineDataRows* . . . 41

B Digital content **42**
B.1 *SLN_old* . . . 42
B.2 *SLN_new* . . . 42
B.3 *Running integration tests in Kentico* . . . 43

1 Introduction

In today's world, testing is not just a complement to the development phase of a software product but its full and inseparable part. Undoubtedly, some parts of the testing process need still be done manually, others can be fully automated. Sometimes the automated tests run periodically in CI[1] to ensure the maximum quality and responsiveness to changes in code.

Automated testing is a cheap way of maintaining the overall quality of a software. Automated tests can not only discover bugs but also make sure that any kind of refactoring does not break the intended functionality of the code.

Whether a company uses TDD[2] or integrate automated tests in their software in other ways, these tests are necessary for any successful long-term project. Without automated tests, the software is prone to bugs without repeated manual re-testing.

It is not always possible to use real objects for the testing and that is why *fake* or *mock* objects are used. These terms are sometimes used as synonyms, some approaches, on the other hand, consider them different.

Different types of tests exist as a way of testing on more levels and different ranges of the SUT[3]. From the smallest - *unit* tests that test the functionality of a single method or unit, through *integration* tests testing different parts correctly working together, all the way to *system* tests that test the system as a whole. This thesis focuses mainly on *unit* and *integration* tests.

While it is easy in theory to differentiate between the types of tests, in real life, it might be more difficult to draw the line. Sometimes a *unit* test testing a single method is not enough as a result of current implementation or design that might require, for example, a database for this method to function correctly. This is only one example of a situation where usually *unit* tests are used but are not enough due to incomplete isolation of this test from external dependencies.

---

1. Continuous integration
2. Test-driven development
3. System under test

The aim of this thesis is to analyze faking objects in Kentico CMS[4] and its insufficiencies when it comes to performing complex SQL operations on Kentico's objects, namely the *JOIN* operation.

The thesis is divided into six chapters, the first of which introduces the reader the context and briefly explains the problem this thesis addresses.

In the second chapter, Kentico's back-end that consists of *Info* and *InfoProvider* objects is described with a sample implementation that allows for an easier understanding of the functionality of these objects.

The third chapter describes *DataQuery* which is Kentico's object-oriented abstraction of the database query for reading data. The chapter further discusses its functionality and current drawbacks of using the more complex *JOIN* operation in the memory.

The fourth chapter outlines automated testing, its importance and the types of tests discussed in this thesis. This chapter also provides a few examples of how to mimic real objects in tests using *fakes* or *mocks*.

The fifth chapter discusses automated testing in Kentico, what types of tests are used in Kentico and also gives a brief description of *NUnit*[5] framework that is used for automated testing in Kentico.

The last, sixth, chapter describes the solution in the form of a new layer of abstraction that is implemented to allow in-memory *JOIN* operation on objects using *LINQ*[6]. This chapter also presents the performance improvement on tests and discusses further extensions or alternative solutions.

4. https://en.wikipedia.org/wiki/Kentico_CMS
5. https://github.com/nunit/nunit
6. .NET Language-Integrated Query

2 Kentico's CMS structure

Kentico is a content management system (CMS) developed by Kentico Software located in Brno, Czech Republic. CMS is a software system that provides website authoring, collaboration, and administration tools designed to allow users with little knowledge of web programming languages or markup languages to create and manage website content[1].

Kentico's functionality covers five areas: content management, e-commerce, social networking, intranet, and online marketing. It utilizes *ASP.NET* web framework and *Microsoft SQL Server* and to its all-in-one solution for digital agencies is sometimes referred as Kentico EMS[1].

2.1 Kentico back-end

All entities in Kentico, such as users or page templates, are represented as objects with corresponding classes. These classes are called `<Object>Info` and `<Object>InfoProvider` where `<Object>` is the name of the entity represented by the given class.

This design pattern is called *Provider model*[2] and was formulated by Microsoft for use in the ASP.NET Starter Kits and formalized in .NET version 2.0[2].

2.1.1 Info class

In Kentico API[3], objects are represented by *Info class* classes. In most cases, the Info class properties (e.g. FirstName of the user object) reflect columns of the database table that stores the particular object. A single instance of an Info class holds the data of one object only, representing one row in the database table.

For demonstration purposes, we will be working with the *User* object represented by *UserInfo* and *UserInfoProvider* classes. The declaration of the UserInfo class is the following:

---

1. Enterprise Marketing Solution
2. https://msdn.microsoft.com/en-us/library/aa479020.aspx
3. Application programming interface

```
public class UserInfo : AbstractInfo<UserInfo>,
   IUserInfo
{
    // Object type
    public const string OBJECT_TYPE =
       PredefinedObjectType.USER;

    // Type information
    public static ObjectTypeInfo TYPEINFO =
       new ObjectTypeInfo(typeof(UserInfoProvider),
       OBJECT_TYPE, ...) { }

    // Properties, constructors, and methods ...
}
```

All the Info classes inherit from the `AbstractInfo<TInfo>` abstract class where `TInfo` is the type of the object we are declaring. Every Kentico's predefined object type that exists in the CMS has the public `OBJECT_TYPE` constant that provides the string representation of the object's type. Also, every Info class has a public static property called `TYPEINFO`.

2.1.2 InfoProvider class

Every Info class has its *InfoProvider* counterpart, which provides methods for managing data - CRUD[4] operations.

Analogically to the *Info* class, the *InfoProvider* inherits from its parent abstract class `AbstractInfoProvider<TInfo, TInfoProvider>`.

The declaration of an InfoProvider class is simple, again using the *User* object type:

```
public class UserInfoProvider :
   AbstractInfoProvider<UserInfo, UserInfoProvider>
{
    // InfoProvider body
}
```

4. CRUD is the acronym for the four basic functions of persistent storage - create, read, update and delete

3 Kentico DataQuery API

3.1 DataQuery

DataQuery is an object-oriented abstraction of the database query for reading data. It eliminates the need for using SQL commands with their parameters, data adapters, connections, etc.

The key aspects of the *DataQuery* are the following:

- support for LINQ and strongly-typed enumeration
- abstraction of the database syntax
- parameter-driven queries
- lazy loading of results
- centralized source point of data
- security

3.2 ObjectQuery

Strongly typed extension of the *DataQuery* is called ***ObjectQuery***. In the Kentico API, all Info Providers return this strongly typed query. With *ObjectQuery*, we can either get the results as `InfoDataSet` through the `TypedResult` property, or enumerate it directly with the `foreach` statement.

3.3 DataQuery concept

Every data query base has a general query behind, with the full name `<class name>.generalselect`. This query is generated on-the-fly for each individual class. This is how this query looks like no matter from which class it comes from:

```
SELECT    ##DISTINCT_COLS## ##TOP_N## ##COLUMNS##
FROM      ##SOURCE##
WHERE     ##WHERE_COND##
GROUP BY  ##GROUP_BY##
HAVING    ##HAVING_COND##
ORDER BY  ##ORDER_BY##
```

For example, to get all users from the database, we can use the following code:

```
var users = UserInfoProvider.GetUsers();
```

The *users* variable will be cast to `DataQuery<UserInfo>` type. To get the generated SQL query, the `users.ToString()` method can be used to return the string representation of the generated query or the `users.Execute()` method to execute the query:

```
SELECT * FROM CMS_User
```

Furthermore, the query can be fine-tuned with *WHERE* condition(s) which can be defined in two ways. Either as a standalone object of the type `WhereCondition`, which can be later passed to the query, or directly at the query level. The resulting *where* condition forms the content of the `##WHERE_COND##` placeholder in the query.

1. Using *WhereCondition* type

   ```
   var where = new WhereCondition()
       .Where("ColumnName", QueryOperator.Equals,
          "SomeValue");
   ```

   This matches the options that the SQL server offers for binary operators and produces the following code:

   ```
   WHERE ColumnName = N'SomeValue'
   ```

2. Using in-line where condition

   ```
   var users = UserInfoProvider.GetUsers()
       .Where("UserName", QueryOperator.Equals,
          "administrator");
   ```

   Using the `users.ToString()` returns the following query:

   ```
   SELECT * FROM CMS_User
   WHERE UserName = @UserName
   ```

   For debugging, testing, or legacy purposes, we can use the `users.ToString(true)` method override that returns the string representation which materializes all the parameters and produces the following output:

   ```
   SELECT * FROM CMS_User
   WHERE UserName = N'administrator'
   ```

Analogically as the *WHERE condition*, we can use:

1. Specific where conditions
   (a) `WhereContains(string columnName, string value)`
   (b) `WhereLike(string columnName, string value)`
   (c) `WhereIn(string columnName, ICollection<T> values)`
   (d) Etc.
2. Order by
   (a) `OrderBy(params string[] columns)` - general one with enum for direction
   (b) `OrderByAscending(params string[] columns)`
   (c) `OrderByDescending(params string[] columns)`
3. `TopN(int topN)`
4. `Columns(List<string> columnNames)`
5. `GroupBy(List<string> columnNames)`
6. `Having(Action<WhereCondition> condition)`
7. `Distinct()`

A more complex example including *GROUP BY* and *HAVING*:

```
var users =
UserInfoProvider.GetUsers()
    .Columns("UserName", "UserDisplayName")
    .AddColumn(
        new CountColumn("UserName").As("UserCount"))
    .GroupBy("UserName", "UserDisplayName")
    .Having(w => w
        .WhereStartsWith("UserDisplayName","CMS "));
```

With the following result:

```
SELECT UserName, UserDisplayName,
   (COUNT(UserName))
AS UserCount
FROM CMS_User
GROUP BY UserName, UserDisplayName
HAVING UserDisplayName LIKE N'CMS %'
```

DataQuery allows us to completely abstract all code from SQL specific syntax which provides high scalability for the future[3].

3.3.1 DataSource property

`DataSource` is a property of *DataQuery* that indicates where the data of the particular query resign. Its general string format is:

```
"Prefix|GUID|DataSetName|TableName"
```

The `Prefix` placeholder for database data is equal to `Database`, and for in-memory cached or faked data (for fake data, see section 5.2) is equal to `Memory`.

When faking data in Kentico's unit tests (subsection 5.1.1) and later executing the following piece of code:

```
var users = UserInfoProvider.GetUsers();
```

the output of the `users.ToString()` is:

```
SELECT * FROM [ExternalSource]
```

where the `[ExternalSource]` indicates that there are faked data in the memory. The `users.DataSource` describing the location of the data is equal to `{CMS.DataEngine.MemoryDataQuerySource}` while the `users.DataSourceName` is equal to (the *GUID* may be different each time the code is executed):

```
"Memory|f0c96682-830b-4cc5-9016-ec3a3eb298af|
    NewDataSet|UserInfo"
```

Getting all the *Info* objects from the *DataQuery* in the form of a `DataSet` is done by the `Result` property.

```
var users = UserInfoProvider.GetUsers().Result;
```

This code will cast the `users` variable to `DataSet` type that contains one `DataTable` with data rows containing `UserInfo` objects. This allows for easy querying of the included data rows by LINQ.

3.3.2 DataSet

The *DataSet* class has been designed as an offline container of data. It consists of a set of data tables, each of which will have a set of data columns and data rows[4].

3.4 DataQuery Join types

JOINs are used to query data from two or more tables, based on a relationship between certain columns in these tables.

Out-of-the-box, Kentico's *DataQuery* supports three types of joins:

INNER JOIN

This query returns all of the records in the left table with matching records from the right table.

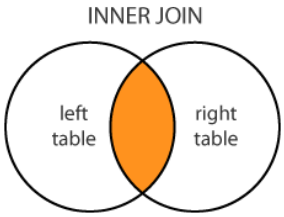


Figure 3.1: Graphical representation of Inner Join

LEFT JOIN & RIGHT JOIN

Left Join returns all of the records in the left table regardless if any of those records have a match in the right table. It will also return all matching records from the right table. Right Join works analogically for the right table[5].

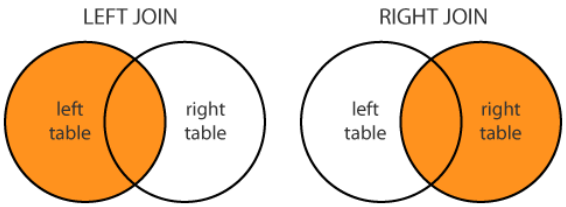


Figure 3.2: Graphical representation of Left and Right Join

The *DataQuery* representation of the join is:

```
SELECT * FROM [LTable] ##JOIN_TYPE## [RTable]
ON [LTable].[LColumn] = [RTable].[RColumn]
```

3.5 Current implementation

As described in section 3.3, the *DataQuery* currently supports the majority of *SQL* statements. The robustness of the *DataQuery* is also the fact that these statements can also be combined and chained to achieve the desired outcome when both working with in-memory objects and retrieving data from a database. As an example, take the following code:

```
var users = UserInfoProvider.GetUsers()
    .Where("UserName", QueryOperator.Like, "a%",
       StringComparison.OrdinalIgnoreCase)
    .Where("UserAge",
       QueryOperator.GreaterOrEquals, 18);
```

This chaining will concatenate the *Where* conditions with **AND** operator, and will return all users whose name starts with *'a'* and their age is greater than or equals to 18.

3.5.1 The Source method

The `Source` method is used to define the source of the query. Its implementation is as follows:

```
public TQuery Source(Action<QuerySource>
   sourceParameters)
{
    var result = GetTypedQuery();
    // Create source instance from current source
    var source = GetSource();
    sourceParameters?.Invoke(source);
    result.QuerySource = source;

    // Include parameters from source condition
    result.IncludeDataParameters(source.Parameters,
       source);

    return result;
}
```

3.6 DataQuery insufficiencies

Despite these features, the *DataQuery* is not capable of performing the *JOIN* operation in the memory. To join two *Info* objects in Kentico's automated tests, these tests must be of type *integration*. To retrieve the result of joining two *Info* objects, we can use the following code example:

```
var users =
    UserInfoProvider.GetUsers()
        .Source(s =>
            s.Join<UserSettingsInfo>("UserID",
               "UserSettingsUserID")
        );
```

This will cast the *users* variable to the type `DataQuery<UserInfo>` which will return `UserInfo` objects joined on `UserSettingsInfo` objects where the `UserID` column of `UserInfo` object is equal to the `UserSettingsUserID` column in `UserSettingsInfo` object. The resulting *SQL* statement:

```
SELECT * FROM [User]
INNER JOIN [UserSettings]
ON [User].[UserID] =
   [UserSettings].[UserSettingsUserID]
```

When invoking the `Join` lambda function in the `Source` method, Kentico first processes the object type passed as the `TObject` type of the `Join<TObject>(...)` function and only then performs the joining of the two *Info* objects. This processing takes place in the following method:

```
public TSource Join<TObject>
    (string leftColumn, string rightColumn,
     JoinTypeEnum joinType = JoinTypeEnum.Inner,
     IWhereCondition additionalCondition = null)
         where TObject : BaseInfo, new()
{
    return Join(new ObjectSource<TObject>(),
       leftColumn, rightColumn,
       additionalCondition, joinType);
}
```

Here, the `new ObjectSource<TObject>()`, the first parameter in the `return` statement, tries to infer the object source. Firstly, it tries the object type's `DataSource` property, and secondly, if the `DataSource` is not defined, it queries the database.

Kentico stores all metadata about its existing classes in the database in the `CMS_Class` table. Before finding the object's source, Kentico has to look up the class schema in this table's `ClassXmlSchema` column in order to know what is the class' schema is and what is the corresponding table name. This table name then becomes the object's source for the automated tests.

When running automated tests inheriting the `IntegrationTests` class, Kentico sets up a connection string to make database access possible. This has to be an already existing Kentico database with data in contrast with the isolated integration tests. On the other hand, when inheriting the `UnitTests` base class in our automated tests, this connection string is, naturally, omitted.

Then trying to join two *Info* objects in unit tests fails as the result of the current implementation that does not set correctly the `DataSource` of the `TObject`'s type. With the `DataSource` being `null` and not having the connection string set up because the inheritance from the `UnitTests` base class, the `Join<TObject>()` method fails with a `NullReferenceExeption` and an error message saying: `"The ConnectionString property has not been initialized."`

4 Test automation

In software testing, *test automation* is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually[6].

4.1 Test automation benefits

The benefits of test automation are plenty. Below are four common ones:

- **Reliable** - Tests perform the same operations precisely each time they are run, therefore eliminating human errors.
- **Fast** - Test execution is faster than done manually.
- **Repeatable** - Once tests are created, they can be run repeatedly with little effort, even at lunchtime or after working hours.
- **Regression Testing** - The intent of regression testing is to ensure that a change, such as a bug fix, did not introduce new faults[7].

4.2 Unit tests

A *unit test* is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterwards. If the assumptions turn out to be wrong, the unit test has failed. A *unit* is a method or function[8].

A good unit test should:

- be automated and repeatable,
- be easy to implement,
- run quickly,
- be consistent in its results,
- have full control of the unit under test,
- be fully isolated (runs independently of other tests).

Unit tests are also tests that do not require any external resources, such as a database or network and do not have any external dependencies, such as time or random number generators.

4.3 Integration tests

Integration tests are more complex as they test if different parts of a system are working together correctly and integration testing aims to uncover interaction and compatibility problems as early as possible[9].

While working with integration tests, external dependencies such as a database or network access do not necessarily have to be available in the of time the test execution. The biggest issue, however, is the speed. When using unit tests that work with data stored in the memory, the reading speed is high which makes unit tests run admittedly fast. Reading from a database or a remote location on the network introduces a delay in hundreds of milliseconds which might add up to a significant time difference.

	Unit tests	Integration tests
Speed	*fast*	*slow*
Scope	*narrow* (testing a function or a method)	*wide* (testing different parts of a system)
External dependencies	*no* (using only the memory as data storage)	*yes* (database, network, local disk, ...)
Isolation	*yes* (runs independently of other tests)	*no* (integrates testing of different parts)
Control over the test	*total* (a well written unit test will always run)	*partial* (external resources availability is unknown until the test is executed)

Table 4.1: Comparison of unit and integration tests

4.4 Mimicking real objects in automated tests

In general, *Mock* objects are used in software testing to simulate software dependencies so that the testing process can be accelerated and the testing scope can be limited to the component under test (instead of going beyond the interface of dependencies and invoke potential bugs relevant to dependencies).

To simulate real dependencies, mock objects typically have the same interface as the objects they mimic. Therefore, the client object remains unaware of whether it is using a real object or a mock object[10].

In other words, a mock object is simply a testing replacement for a real-world object. A number of situations can come up where mock objects can help us:

- The real object is difficult to set up, requiring a certain file system, database, or network environment.
- The real object has behaviour that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to confirm that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

Using mock objects, we can get around all of these problems. The three key steps to using mock objects for testing are as follows:

1. Use an interface to describe the relevant methods on the object.
2. Implement the interface for production code.
3. Implement the interface in a mock object for testing.

The code under test refers to an object only by its interface or base class, so it can remain unaware of whether it is using the real object or the mock[11].

4.4.1 Mocks vs. Fakes vs. Stubs

The term *Mock Objects* has become a popular one to describe special case objects that mimic real objects for testing. Most language environments now have frameworks that make it easy to create mock objects. What's often not realized, however, is that mock objects are but one form of special case test object[12]. Terms used to describe objects that simulate real objects are as follows:

- **Dummy** objects are passed around but never actually used. Usually, they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually, take some shortcut which makes them not suitable for production (an in-memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Generally, in Kentico, the term *Fake* objects is used to describe *Info* and *InfoProvider* objects that are used in unit tests. Faking objects in Kentico is described in section 5.2.

5 Automated testing in Kentico

5.1 Types of tests in Kentico

Kentico uses *NUnit* for automated testing. NUnit is an open source unit-testing framework for Microsoft .NET platform and its *assert model* is used in all Kentico's automated tests. More about NUnit framework can be found in section 5.3.

Kentico's automated tests are divided into three categories: *Unit tests*, *Integration tests*, and *Isolated integration tests*[13].

5.1.1 Unit tests

Unit tests are able to run without external resources, such as a database, and execute much faster than the other types of automated tests. When working with *Info* and *InfoProvider* objects, we can avoid accessing the database by using Kentico's fake objects by using `Fake<TInfo, TInfoProvider>()` method to fake the desired objects. Any unit test containing class must inherit from the `UnitTests` base class.

5.1.2 Integration tests

Integration tests can access a database provided by a connection string. These tests are used when data read is needed from the database which makes them significantly slower than unit tests. Integration test class inherits the `IntegrationTests` base class.

5.1.3 Isolated integration tests

Isolated integration tests automatically create their own database before the test execution and clean up the database after the test is finished. This type of tests is the slowest of the three types of automated tests and their base class is `IsolatedIntegrationTests`.

5.2 Faking Info and InfoProvider objects in unit tests

To create fake data for unit tests, the `Fake()` method must be called from the *UnitTests* base class (must be inherited by all unit test classes using the *CMS.Tests* library). The method prepares a faked instance of a specific *InfoProvider* class with data. Then use the given InfoProvider's methods to *get*, *create* or *update* data in unit tests.

```
[TestFixture]
public class MyUnitTests : UnitTests
{
    [SetUp]
    public void SetUp()
    {
        // Prepares faked data
        Fake<UserInfo, UserInfoProvider>()
            .WithData(
                new UserInfo
                {
                    UserID = 1,
                    UserName = "FakeUser"
                }
            );
    }

    [Test]
    public void MyTest()
    {
        // Calls a provider method to get user data
        var users = UserInfoProvider.GetUsers();
    }
}
```

In the example, *UserInfoProvider* is faked with specific data. Calling *UserInfoProvider* methods to get *UserInfo* objects within tests then returns the faked data instead of accessing the database. The code in the example behaves as if there was a single user named *FakeUser* in the database[14].

There is **no need to clean up the faked data before or after running the tests**. Tests inherited from the *UnitTests* base class automatically reset all faked data upon initialization and cleanup.

5.3 NUnit framework

NUnit is a unit-testing framework for all .NET languages. Initially ported from *JUnit* (unit-testing framework for Java), the current production release, version 3, has been completely rewritten with many new features and support for a wide range of .NET platforms.

5.3.1 Nunit attributes and life cycle

`[TestFixture]` attribute

This attribute marks a class that contains tests. This attribute can also be used for declaring a base class for other (parametrized) test fixture derived classes to inherit. Test fixtures may take constructor arguments.

`[OneTimeSetUp]` and `[OneTimeTearDown]` attributes

These attributes identify methods that are called once prior to and after executing any of the tests in a fixture. If a OneTimeSetUp method fails or throws an exception, none of the tests in the fixture are executed and a failure or error is reported. If any of the methods marked with *OneTimeSetUp* attribute runs, the *OneTimeTearDown* method is guaranteed to run.

`[SetUp]` and `[TearDown]` attributes

These attributes are used inside a *TestFixture* to provide a common set of functions that are performed just before and just after each test method is called.

Any methods marked with one of the attributes: `OneTimeSetUp`, `OneTimeTearDown`, `SetUp`, or `TearDown` may be either static or instance methods that can be defined multiple times. Normally, multiple methods are only defined at different levels of an inheritance hierarchy and are run in this order.

`[Test]` and `[TestCase()]` attributes

The *Test* and *TestCase* attributes are the way of marking a method inside a *TestFixture* class as a test.

The *Test* attribute marks simple (non-parameterized) tests but may also be applied to parameterized tests without causing any extra test cases to be generated.

TestCase attribute serves the dual purpose of marking a method with parameters as a test method and providing in-line data to be used when invoking that method.

`[TestCaseSource]` attribute

TestCaseSource attribute is used on a parameterized test method to identify the source from which the required arguments will be provided. The attribute additionally identifies the method as a test method. The data is kept separate from the test itself and may be used by multiple test methods.

This attribute can be used in two distinct forms. One to identify the test case data within the same class - usage `[TestCaseSource(string sourceName)]` or loading the test case data from a different class - usage `[TestCaseSource(Type sourceType, string sourceName)]`[15].

5.3.2 Assertions

Assertions are central to unit testing in any of the xUnit frameworks, and NUnit is no exception. NUnit provides a rich set of assertions as static methods of the `Assert` class.

An assertion method compares the actual value returned by a test to the expected value, and throws an *AssertionException* if the comparison test fails[16]. If a test contains multiple assertions, any that follow the one that failed will not be executed. For this reason, the best practice is one assertion per test.

Each method may be called without a message, with a simple text message or with a message and arguments. In the last case, the message is formatted using the provided text and arguments.

5.3.3 Constraint-based assert Model

The constraint-based Assert model uses a single method of the `Assert` class for all assertions. The logic necessary to carry out each assertion is embedded in the constraint object passed as the second parameter to that method.

Using this model, all assertions are made using one of the forms of the `Assert.That()` method, which has a number of overloads:

```
Assert.That(object actual, IConstraint constraint)
Assert.That(object actual, IConstraint constraint,
   string message)
Assert.That(object actual, IConstraint constraint,
   string message, object[] params)

Assert.That(bool condition);
Assert.That(bool condition, string message);
Assert.That(bool condition, string message,
   object[] parms);
```

For overloads taking a constraint, the argument must be an object implementing the `IConstraint` interface, which supports performing a test on an actual value and generating appropriate messages[17].

Assume there is a method `GetHypotenuseLength(int a, int b)` that takes two integers as parameters, which represent the legs of a right triangle and calculates the length of the hypotenuse using the Pythagorean theorem. Example of a test case testing this method would look the following:

```
[Test]
public void TestExample()
{
    var length = GetHypotenuseLength(3, 4);

    Assert.That(length, Is.EqualTo(5));
}
```

6 Solution and performance measurements

6.1 Solution

To be able to run unit tests containing the *Join* operation on two or more *Info* objects, the following steps need to be undertaken:

1. Faking the class structure in unit tests to make sure that the unit tests will not query the database.
2. Implement logic to the `Source` method that will prepare the ground to join objects in the memory.
3. Implement a new layer of abstraction that will have the data sources of both *Info* objects available at the same time.
4. Implement the logic of joining of the two *Info* objects.
5. Setting the `DataSource` property of the query to the result of previously joined objects.

6.1.1 Faking the class structure

In order to avoid querying the database when inferring the object's type (as discussed in section 3.6), the `DataClassInfo` (code equivalent to the `CMS_Class` table) has to be faked in the `UnitTests` base class. To achieve this the following line of code needs to be added to the `Init` method in the `UnitTests` class:

```
[OneTimeSetUp]
public void Init()
{
    Fake<DataClassInfo, DataClassInfoProvider>()
        .WithData();
}
```

This will ensure that whenever any piece of code tries to infer an object's type during the run of the tests, it will look into the memory for the faked structure instead of querying the database.

6.1.2 Implementing logic to the `Source` method

The `Source` method (see subsection 3.5.1) on the line

```
var source = GetSource();
```

gets the `DataSource` property (subsection 3.3.1) of the *DataQuery*. If Kentico runs automated unit tests, the type of `DataSource` is set to `MemoryDataQuerySource` provided that the *DataQuery* runs in the context of in-memory faked data. We can use this information to create a new layer of abstraction that will allow to join objects in the memory. The following entry is added to the `Source` method:

```
if (DataSource is MemoryDataQuerySource)
{
    return MemorySource(sourceParameters, result,
       source);
}
```

The `MemorySource` method is very similar to the `Source` method but it works with a new memory source that is the new layer of abstraction described in subsection 6.1.3. The implementation of the `MemorySource` method can be found in section A.1.

6.1.3 Implementing a new layer of abstraction

A new layer of abstraction is needed in order to have the data sources of both objects we are joining available. Since all logic that is required for a successful join takes place in the memory, the new class will be called `MemorySource` (not to be confused with the method) that will inherit from the `ObjectSource` class. The key concept of the `MemorySource` class is that it has three properties:

- `LeftDataSource` that is the data source of the first (or left) object of the join,
- `RightDataSource` that is the data source of the second (or right) object of the join,
- `ResultDataSource` that is the result of the join operation over the left and right objects.

The implementation if the `MemorySource` class can be found in section A.2. The `Join` method in the `MemorySource` has three key functionalities:

1. Setting the `LeftDataSource` in the constructor,
2. Setting the `RightDataSource` and the `ResultDataSource` in the method `PrepareDataSources` (see section A.3), and
3. Calling the `Join` method of the parent class to prepare the correct `DataQuery`.

The `PrepareDataSources<TObject>` method determines the object source from the `TObject` generic parameter. As we have already faked the `DataClassStructure` in subsection 6.1.1, the code will not end with an exception. Next, the code gets the `InfoProvider` object based on the object source.

Every Kentico's `InfoProvider` object implements the interface `ITestableProvider` which contains the `DataSource` property that is in our case the `RightDataSource`.

Lastly, the result of the `InMemoryJoin.Join()` method that handles the logic of the joining of the two data sources is set to the `ResultDataSource` property of this class which is later assigned to the resulting data source in the `MemorySource` method.

6.1.4 Implementing the logic of joining

Both data sources of the objects to be joined contain the faked data in a `DataSet`. Each of these datasets contain exactly one `DataTable` at index *0* and the data rows contained in these tables represent the faked data. One `DataRow` equals one object created with the `new` keyword within the `WithData( new { ... } )` method of the fakes.

Inner Join

The logic of joining is done by LINQ and the implementation can be found in section A.4. This method returns all the data rows from both tables that have a match on `leftRow.Field<int?>(leftColumn)` equals `rightRow.Field<int?>(rightColumn)` fields.

Left Join

This method's implementation can be found in section A.5 where the `GropuJoin` returns every row from the left table with a list of rows from the right table that have a match on the desired columns passed as `leftColumn` and `rightColumn` parameters. Furthermore, the `SelectMany` creates tuples of every left row from the list of matching right rows and combines them into one. The `DefaultIfEmpty()` method ensures that if the left row has no match in the right table, `null` will be returned instead of skipping the result.

The right join works in the same manner, only with the difference that it will return every row from the right table with a list of matching rows from the left table[18].

All of these methods have a reference to the `CombineDataRows()` method (see section A.6). This method takes two data rows and combines them into one. Since no data row can exist without a parent table with its own schema, a new data table is created for the purposes of concatenating the data rows. The resulting data table is a concatenation of the columns of the left and right tables with a simple duplicity check.

6.1.5 Setting the `DataSource` property

The previously created data rows that are the result of the join operation are then encapsulated into a new `DataTable` which is further encapsulated into a new `DataSet` object that is passed into the constructor of the `MemoryDataQuerySource`.

This new memory source is then set as the `ResultDataSource` property of the abstraction described in subsection 6.1.3.

```
result.DataSource = memorySource.ResultDataSource;
```

The full implementation with all the missing helper methods can be found in the digital appendix to this thesis. The missing methods are omitted from the text as their implementation is not required to understand the context but are absolutely crucial for the final prototype to work correctly. A brief description how to run the *unit* or *integration* tests can be found in Appendix B.

6.2 Performance measurements

Measurement of performance was carried out on the local workstation with parameters:

- OS: Windows 7 Enterprise SP1
- CPU: Intel(R) Core(TM) i5-4690 (quad core, 6 MB, 3.50 GHz)
- RAM: 2 x 8 GB, 1600 MHz, DDR3, Non-ECC

Each measurement was performed 10 times in different settings - in memory, using a locally installed database, using a database hosted on a server within Kentico intranet. Using a database hosted on *Microsoft Azure* cloud service was excluded as Kentico does not run its own tests in the cloud.

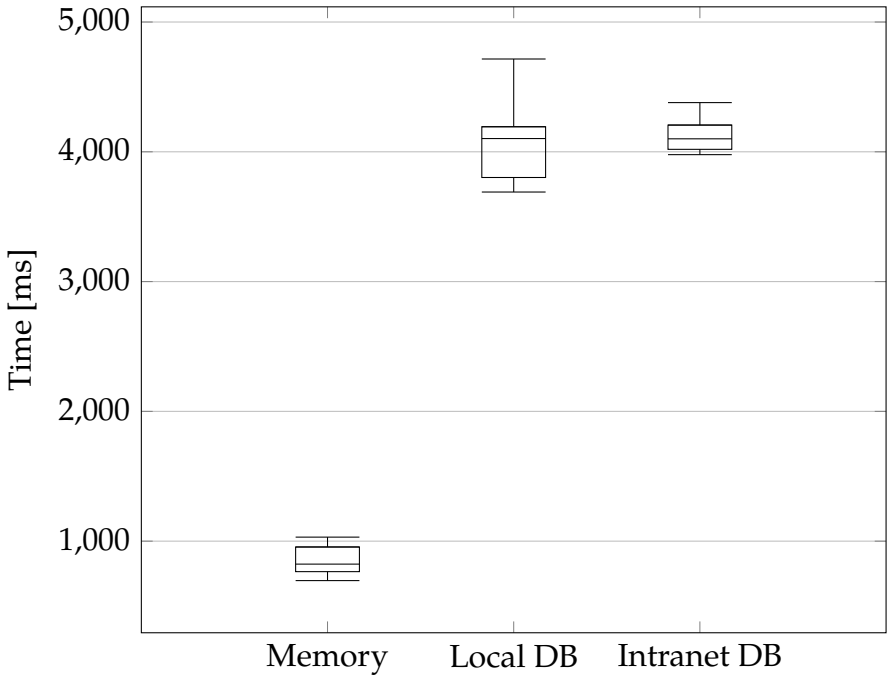
The measurement was carried out by the *dotTrace*[1] performance profiler and the measured times are in *milliseconds* with the average being rounded to the nearest whole number. On the next page follow the measurements of the tests.

1. https://www.jetbrains.com/profiler/

6.2.1 Running times of 100 automated tests

Number	Memory	Local DB	Intranet DB
1.	818	4 715	4 130
2.	1 031	3 748	4 031
3.	709	4 160	4 229
4.	779	3 690	4 146
5.	989	3 802	4 206
6.	828	4 193	3 978
7.	1 027	4 005	4 069
8.	696	4 139	4 379
9.	765	4 066	4 019
10.	955	4 336	3 985
Average	**860**	**4 085**	**4 117**

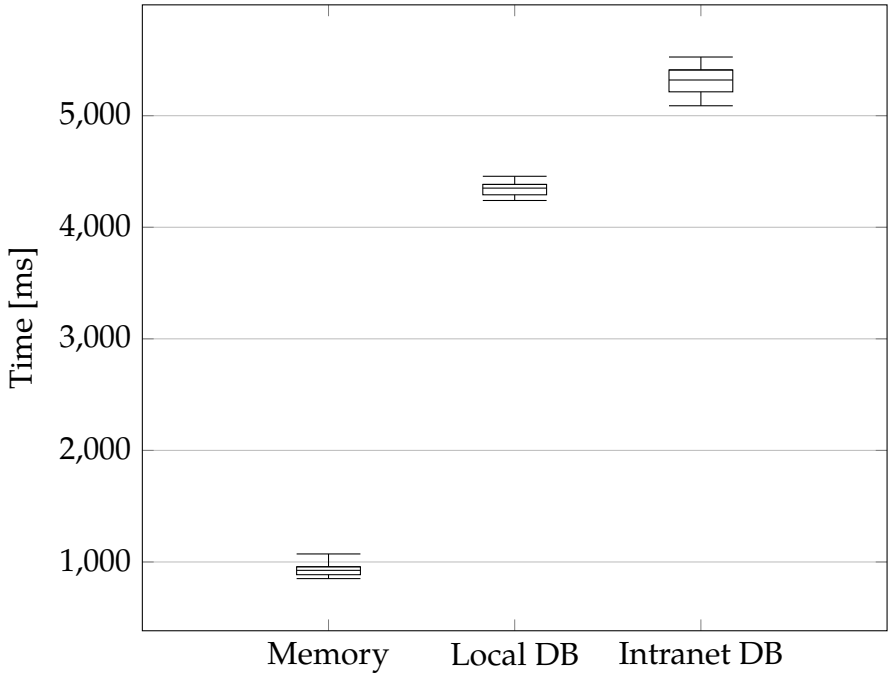
Table 6.1: Running times of 100 automated tests in *milliseconds*



6.2.2 Running times of 500 automated tests

Number	Memory	Local DB	Intranet DB
1.	1 072	4 317	5 330
2.	909	4 376	5 526
3.	889	4 384	5 452
4.	856	4 282	5 214
5.	941	4 291	5 310
6.	851	4 346	5 400
7.	957	4 355	5 089
8.	886	4 457	5 410
9.	950	4 392	5 193
10.	976	4 240	5 275
Average	**929**	**4 344**	**5 320**

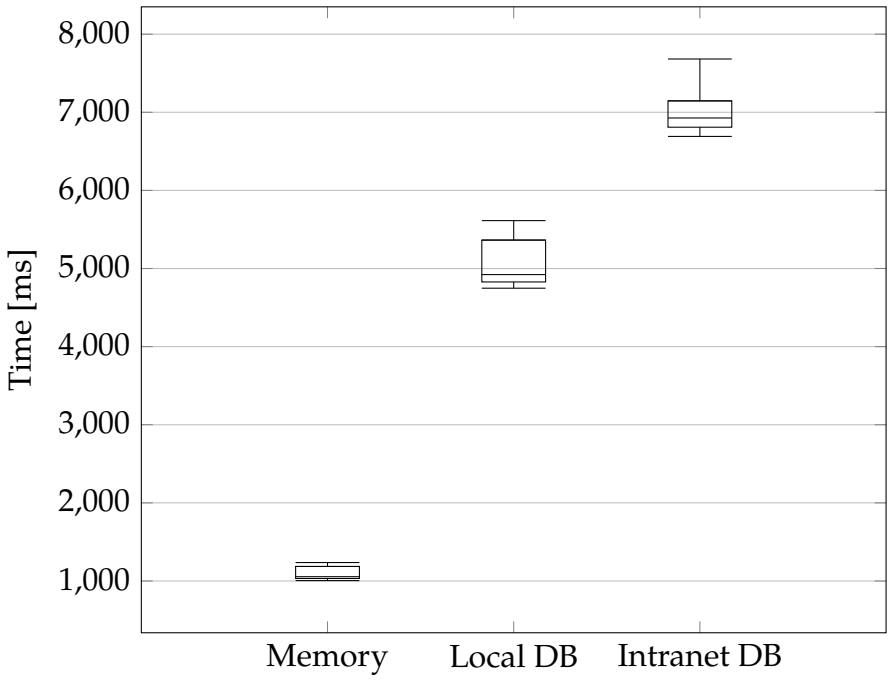
Table 6.2: Running times of 500 automated tests in *milliseconds*



6.2.3 Running times of 1000 automated tests

Number	Memory	Local DB	Intranet DB
1.	1 016	5 613	6 953
2.	1 201	4 828	7 146
3.	1 113	5 364	6 809
4.	1 005	4 859	6 691
5.	1 062	4 748	7 144
6.	1 236	4 934	6 900
7.	1 189	5 188	7 682
8.	1 096	5 538	6 731
9.	1 032	4 909	7 254
10.	1 127	4 759	6 879
Average	**1 107**	**5 074**	**7 019**

Table 6.3: Running times of 1000 automated tests in *milliseconds*



6.2.4 Conclusion

When using memory as a data storage for the tests instead of a database the performance gain is significant. The automated tests (all types) run after every check-in to Kentico's code base in CI. In addition, these tests are also periodically scheduled to run every night independently of code changes.

Running 100 tests

The performance gain of running 100 tests in-memory in comparison with a locally installed database is around *475%* and around *479%* when running tests against a database.

Running 500 tests

500 automated tests in-memory gains approximately *468%* on performance using a local database and *573%* when having an intranet database.

Running 1000 tests

If having 1000 automated tests, the running time improves around *458%* compared to a locally installed database and *634%* compared the intranet database.

6.2.5 Testing hidden joins

Apart from the performance gain, the most significant benefit of this solution is the ability of testing *hidden joins*. This means testing parts of the system implementing the *Join* method without explicitly having to do the necessary joining of objects in the tests.

6.3 Extensions and alternative solutions

6.3.1 Extensions

The current implementation as described in subsection 6.1.4 joins the objects by fields `leftColumn` and `rightColumn` that are passed to the `Join` method as parameters.

Supporting other types

The fields `leftColumn` and `rightColumn` are of type `<int?>` (nullable integer)[19] that is the most commonly used type for joins. This implementation can be further extended with generics to support joining on other types, e.g. `string` is also used every now and then.

Including a *where* condition

The `Join` method can also be extended with one more parameter which would represent a *where* condition to filter out unwanted results right away without adding them to the output and using the `Where()` method afterwards. The predicate would be represented by a parameter with the type of `Func<TSource, bool>`. This parameter already resides in the code for easier implementation in the future.

Leverage multi-threading

The return type and the type of the first two parameters of the `Join` method could be changed from `IEnumerable<DataRow>` to a parallel type `ParallelQuery<DataRow>` and using the `AsParallel()` method to run the joining operation in multiple threads using *Parallel LINQ*[20].

Ordering the results

When using the `ParallelQuery<T>` the order of the results is may vary from time to time depending on how many resources has which thread available. To ensure that the result is always in the same order the `AsOrdered()` method is chained after the `AsParallel()` method.

6.3.2 Alternative solutions

The only viable alternative solution with the current implementation of the *DataQuery* would be getting all the items of both *Info* objects' data sets through their `Result` property and perform the join operation on them. Example of such code using the *UserInfo* and *UserSettingsInfo* follows:

```
var users = UserInfoProvider
                .GetUsers().Result;

var userSettings = UserSettingsInfoProvider
                       .GetUserSettings().Result;

var result = Join(users, userSettings,
                  "UserID", "UserSettingsUserID");
```

This solution would still need custom joining logic with the use of LINQ (similar to the current implementation). The resulting `DataSet` would contain the joined objects but it would lose the *DataQuery* abstraction encapsulating the result.

This effectively means that any other modification of the result would require the use of methods over the `DataSet` and the containing `DataTable` or `DataRow` objects, or additional custom logic that would need further implementation.

7 Conclusion

The aim of this thesis was to analyze and extend the current faking of *Info* and *InfoProvider* objects in Kentico's automated tests. The analysis was concluded on Kentico's *DataQuery* API that is an object-oriented abstraction of the database query for reading data.

Chapter three described *DataQuery*, its functionality and a brief listing of methods that make up the *DataQuery*. This chapter also focused on biggest current deficiency which is *DataQuery's* inability to perform *JOIN* operations when inheriting the `UnitTests` base class, meaning running all operations in the memory.

Chapter five described the types of tests existing in Kentico and further discussed how *faking* of objects in Kentico's automated tests works. Chapter two and four contained theoretical context for chapters three and five respectively.

Chapter six, the most extensive, describes the prototype solution that was used to implement a new layer of abstraction that provides functionality to join two objects in the memory.

This solution enabled to have the data sources of both left and right objects available at the same time and performing the join on them using *LINQ* without losing the *DataQuery* abstraction on the result.

This solution improves on performance as there is no need for a database to perform the joining of the two objects. The performance measurements are available in chapter six as a demonstration of the speed of the tests running in the memory compared to tests running against a database.

Another acquisition of this prototype is allowing of testing methods that in their implementation join two or more objects. In this way, not only explicit joins in the tests can be tested but a whole lot more code containing these *hidden joins*.

Furthermore, chapter six contains also a list of possible future extensions not just to improve the functionality of the implemented join logic but also a suggestion how to further improve the performance using parallelism.

In addition, an alternative solution is briefly discussed in this chapter with stated arguments how it can work and why current prototype is not using this approach.

Bibliography

1. MAUTHE Andreas; THOMAS, Peter. *Professional Content Management Systems: Handling Digital Media Assets*. John Wiley & Sons, 2004. ISBN 978-0-470-85542-3.
2. EVJEN Bill; HANSELMAN, Scott. *Professional ASP.NET 4 in C# and VB*. 2nd ed. Wrox, 2010. ISBN 978-0-470-50220-4.
3. *Kentico 8 Technology - DataQuery API* [online]. Brno: HEJTMÁNEK, Martin, 2014 [visited on 2017-09-25]. Available from: `https://devnet.kentico.com/articles/kentico-8-technology-dataquery-api`.
4. ROBINSON Simon; ALLEN, K. Scott. *Professional C#*. 2nd ed. Wrox, 2002. ISBN 978-0-7645-4398-2.
5. *SQL JOIN* [online]. Data & Object Factory, LLC, 2017 [visited on 2017-10-05]. Available from: `http://www.dofactory.com/sql/join`.
6. KOLAWA Adam; HUIZINGA, Dorota. *Automated Defect Prevention: Best Practices in Software Management*. 1st ed. Wiley-IEEE Computer Society Press, 2007. ISBN 0-470-04212-5.
7. ZHAN, Zhimin. *Practical Web Test Automation*. 2nd ed. Lean Publishing, 2017. ISBN 9781505882896.
8. OSHEROVE, Roy. *The Art of Unit Testing*. 2nd ed. Manning Publications Co, 2014. ISBN 978-1-617-29089-3.
9. PEZZÉ Mauro; YOUNG, Michal. *Software Testing and Analysis: Process, Principles, and Techniques*. 1st ed. John Wiley & Sons, 2008. ISBN 978-0-471-45593-6.
10. WANG Xiaoyin; MOSTAFA, Shaikh. An Empirical Study on the Usage of Mocking Frameworks in Software Testing [online] [visited on 2017-12-01]. Available from: `http://ieeexplore.ieee.org/document/6958396/`.
11. HUNT Andy; THOMAS, Dave. *Pragmatic Unit Testing in C# with NUnit*. 2nd ed. The Pragmatic Bookshelf, 2007. ISBN 978-0-9776166-7-1.
12. *Mocks Aren't Stubs* [online]. FOWLER, Martin, 2007 [visited on 2017-12-04]. Available from: `https://martinfowler.com/articles/mocksArentStubs.html`.

13. *Writing automated tests* [online]. Kentico Software, 2016 [visited on 2017-12-05]. Available from: https://docs.kentico.com/k10/custom-development/writing-automated-tests.

14. *Faking Info and Provider objects in unit tests* [online]. Kentico Software, s.r.o., 2016 [visited on 2017-12-04]. Available from: https://docs.kentico.com/k10/custom-development/writing-automated-tests/faking-info-and-provider-objects-in-unit-tests.

15. *NUnit Documentation* [online]. POOLE, Charlie, 2016 [visited on 2017-09-28]. Available from: https://github.com/nunit/docs/wiki/NUnit-Documentation.

16. GARD, Navneesh. *Test Automation using Selenium WebDriver with Java: Step by Step Guide*. AdactIn Group Pty Ltd., 2014. ISBN 978-0-9922935-1-2.

17. *Constraint-Based Assert Model* [online]. POOLE, Charlie; PROUSE, Rob, 2017 [visited on 2017-09-28]. Available from: http://nunit.org/docs/2.4/constraintModel.html.

18. *C# – LINQ – How to Create a Left Outer Join* [online]. LANDRY, Chris, 2013 [visited on 2017-12-05]. Available from: https://topherlandry.wordpress.com/2013/06/25/c-linq-how-to-create-a-left-outer-join/.

19. *Nullable Types (C# Programming Guide)* [online]. Microsoft Corporation, 2017 [visited on 2017-12-05]. Available from: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/.

20. MAGENNIS, Troy. *LINQ TO OBJECTS USING C# 4.0*. 1st ed. Addison-Wesley, 2010. ISBN 978-0-321-63700-0.

A Implementations

A.1 MemorySource method

```
/// <summary>
/// Defines a memory source of data.
/// </summary>
private TQuery MemorySource(Action<QuerySource>
   sourceParameters, TQuery result, QuerySource
   source)
{
    // Create a new memory source
    var memorySource =
        new MemorySource(DataSource, source);

    // Invoke the lambda function
    sourceParameters?.Invoke(memorySource);

    // Set the appropriate sources
    result.QuerySource = memorySource;
    result.DataSource =
       memorySource.ResultDataSource;

    // Include parameters from source condition
    result.IncludeDataParameters(
    memorySource.Parameters,
        memorySource);

    return result;
}
```

A.2 MemorySource class

```
public class MemorySource : QuerySource
{
    private DataQuerySource LeftDataSource { get; }
    private DataQuerySource RightDataSource;

    public DataQuerySource ResultDataSource;

    public MemorySource(DataQuerySource dataSource,
       QuerySource querySource)
        : base(querySource)
    {
        LeftDataSource = dataSource;
    }

    /// <summary>
    /// Performs a join on the given sources.
    /// </summary>
    public override QuerySource Join<TObject>(
        string leftColumn,
        string rightColumn,
        JoinTypeEnum joinType = JoinTypeEnum.Inner,
        IWhereCondition additionalCondition = null)
    {
        ObjectSource<TObject> objectSource;

        // Prepare and set the right and result
           data source
        PrepareDataSources(out objectSource,
           leftColumn, rightColumn,
           additionalCondition, joinType);

        // Encapsulate the result into DataQuery
        return Join(objectSource, leftColumn,
           rightColumn, additionalCondition,
           joinType);
    }
}
```

A.3 PrepareDataSources

```
private void PrepareDataSources<TObject>(
    out ObjectSource<TObject> objectSource,
    string leftColumn, string rightColumn,
    WhereCondition additionalCondition = null,
    JoinTypeEnum joinType = JoinTypeEnum.Inner)
            where TObject : BaseInfo, new()
{
    // Create a new object source from <TObject>
    objectSource = new ObjectSource<TObject>();
    objectSource.SourceExpression =
       GetSourceExpression(new
       QueryDataParameters());

    // Get the InfoProvider of the right object
    var provider = InfoProviderLoader
                   .GetInfoProvider(
                       objectSource.ObjectType)
                           as ITestableProvider;
    if (provider == null)
    {
        throw new InvalidOperationException($"Info
           provider for object type
           '{objectSource.ObjectType}' does not
           implement
           '{typeof(ITestableProvider).FullName}'.");
    }

    // Set the right data source
    RightDataSource = provider.DataSource;

    // Join the two data sources
    ResultDataSource =
       InMemoryJoin.Join(LeftDataSource,
       RightDataSource, leftColumn, rightColumn,
       additionalCondition, joinType);
}
```

A.4 Inner join

```
public IEnumerable<DataRow> Join(
        IEnumerable<DataRow> leftTable,
        IEnumerable<DataRow> rightTable,
        string leftColumn,
        string rightColumn)
{
    return leftTable
    .Join(
        rightTable,
        leftRow =>
            leftRow.Field<int?>(leftColumn),
        rightRow =>
            rightRow.Field<int?>(rightColumn),
        CombineDataRows);
}
```

A.5 Left join

```
public IEnumerable<DataRow> LeftOuterJoin(
    IEnumerable<DataRow> leftTable,
    IEnumerable<DataRow> rightTable,
    string leftColumn, string rightColumn)
{
    return leftTable.GroupJoin(
        rightTable,
        leftRow =>
            leftRow.Field<int?>(leftColumn),
        rightRow =>
            rightRow.Field<int?>(rightColumn),
        (leftRow, matchingRightRows) => new
        {
            leftRow,
            matchingRightRows
        })
        .SelectMany(
            matchinPairOfRows =>
               matchinPairOfRows.matchingRightRows
                                .DefaultIfEmpty(),
            (leftRowWithMatchingRightRows,
               rightRow) =>
            CombineDataRows(
                leftRowWithMatchingRightRows.leftRow,
                rightRow)
        );
}
```

A.6 CombineDataRows

```
public DataRow CombineDataRows(DataRow leftRow,
    DataRow rightRow)
{
    // Check if the left row has a match
    if (rightRow == null)
    {
        return leftRow;
    }

    // Combine the columns from the rows
    var fields =
       leftRow.ItemArray.Concat(rightRow.ItemArray)
                 .ToArray();

    // Create the target table
    var targetTable = CreateTargetTable(
        leftRow.Table,
        rightRow.Table);

    // Add the fields to the target table
    targetTable.Rows.Add(fields);

    return targetTable.Rows[0];
}
```

B Digital content

The digital appendix to this thesis (*SLN.zip*) file contains two folders: *SLN_old* and *SLN_new*. Both of these contain Kentico's `Tests` project under name **CMSTests** and a custom project **Tests**. To run any of the projects, copy the contents of the folder to a local disk, open the project in *Visual Studio* and build the solution. In both cases, make sure that the restored *NuGet* packages are in the correct version and from the correct source as described in each corresponding section.

B.1 SLN_old

The *SLN_old* contains a test project called **Tests** inheriting from the `UnitTests` to demonstrate the behaviour of the unit tests before the implementation of the solution.

Building this solution should restore the *NuGet* packages *Kentico.Libraries* and *Kentico.Libraries.Dependencies* in version *10.0.0* for the projects and the *NUnit* package in the version *3.2.0* from *nuget.org*[1].

B.2 SLN_new

SLN_new folder's content is similar to the old one, except it contains two extra folders:

- *Kentico_nuget*
- *Code_diff*

To ensure the correct run of the tests in this solution, **it is absolutely vital that the project references the *Kentico.Libraries* and *Kentico.Libraries.Dependencies* in the version *11.0.0-B1709081005-2341* from the *Kentico_nuget* folder** and NUnit *3.9.0* from *nuget.org*. **When managing the NuGet packages for this solution, the *Include prerelease* option must be checked in *Visual Studio*.**

The *Code_diff* folder contains independent `.cs` files and screenshots with all the changes in code from the solution including all the helper methods omitted from chapter 6.

1. https://www.nuget.org/

B.3 Running integration tests in Kentico

To run integration tests in Kentico, a database with data is required. Obtaining Kentico with a database with sample data can be achieved by downloading Kentico from the official web page and installing on the local machine.

Downloading Kentico from *https://www.kentico.com/download-demo/* will let the user choose from either a free license or a trial version. Creating an integration test after the installation can be done in just a few minutes by taking the following steps:

1. Create a *Tests* project.

2. Create an integration test (the code from *SLN_new* can be used).

Both steps are described in Kentico's official documentation on the following pages:

1. Creating a tests project:
   https://docs.kentico.com/k10/custom-development/writing-automated-tests/creating-automated-test-projects

2. Creating an integration test:
   https://docs.kentico.com/k10/custom-development/writing-automated-tests/creating-integration-tests-with-a-connection-string