FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	ORM Library for Neo4j Graph Database in .NET Framework
Student:	Bc. Tomáš Starý
Supervisor:	Ing. Marek Skotnica
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The main goal of this work should be to analyze, design, and create an open-source object-relational mapping library (ORM) for a graph database Neo4J in the .NET framework. This solution should enable users to create queries using LINQ.

Steps to take:
- Review EntityFrameworkCore, Neo4j, and Cypher libraries
- Review existing approaches to ORM libraries for graph databases
- Design an ORM library for Neo4J in the .NET framework
- Implement an open-source proof-of-concept of the designed library
- Use test-driven development approach to test the ORM
- Evaluate the results and propose the steps to make the open-source project sustainable for other developers

---

Electronically approved by Ing. Michal Valenta, Ph.D. on 15 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

ORM Library for Neo4j Graph Database in .NET Framework

Bc. Tomáš Starý

Department of software engineering

Supervisor: Ing. Marek Skotnica

April 26, 2022

Acknowledgements

My thanks go to my supervisor Ing. Marek Skotnica, for his support and guidance. I would also like to thank everyone I bothered with the subject of this thesis — thank you, my rubber ducks.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on April 26, 2022 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Starý, Tomáš. *ORM Library for Neo4j Graph Database in .NET Framework*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022. Also available from: ⟨`https://github.com/TomStary/masters-thesis`⟩.

Abstrakt

Tato diplomová práce se zabývá návrhem a implemetencí knihovny, jež vzájemně mapuje grafovou databázi a objekty .NET platformy. Dva hlavní cíle knihovny jsou schopnost mapovat objekty do databáze a schopnost mapovat dotazy z LINQ do Cypher dotazů a jejich výsledky do objektů. Na základě analýzy grafovách databazí, Entity Frameworku a podobných řešení, byl vytvořen návrh knihovny. Knihovna je nápsána v jazyce C# jakožto ověření konceptu a má tedy omezenou funkčnost, zdrojové kódy jsou k dispozici na GitHubu. V závěru je zhodnocen výsledek implementace a jsou předloženy návrhy na další vývoj.

Klíčová slova Entity Framework, grafové databáze, ORM, OGM, Neo4j, .NET, .NET core

Abstract

This master's thesis aims to design and implement an object-graph mapper library for the .NET platform. The two primary goals for the library are the ability to map objects to the database and the ability to map queries from LINQ to Cypher queries and their result to the object. Based on the analysis of the graph database, Entity Framework and the similar solutions, the design for the library was decided. The library is implemented in C# as proof-of-concept with limited functionality, and the source code is available on GitHub. At the end of the thesis, the implementation is evaluated, and the next steps for development are proposed.

Keywords Entity Framework, graph databases, Neo4j, ORM, OGM, .NET, .NET core

Contents

List of Figures

List of Tables

List of source codes

Introduction

Graphs are everywhere

In today's world, everything is highly connected. Graphs are an easy way to describe and visualize these relations and can help to see connections we would not otherwise catch.

We do not use graphs to store data in most of our applications today, mainly because the first applications were made to take paper forms into the digital world. To keep these paper forms in digital format, we use relational databases.

Since then, relational databases have been the go-to for every developer when creating a new application. Using a relational database is helpful for several reasons, mainly because the development cost is lower than using new technologies, and everyone is familiar with this type of DBMS. However, the lower development price benefits are diminished in today's world by the numerous problems with using relational databases. For example, lengthy searches for specific highly connected data and more complex relations are the main reason why NoSQL databases have been gaining so many tractions for the last decade.

The graph databases are made explicitly with relationships in mind. Every vertex can have its connection with another one or even itself. This kind of connection is not possible with a relational database where connecting two rows means creating a relationship between two tables.

What can graph databases offer?

We have discussed the problem with highly connected data and long searches, but how does the graph database solve this issue? The key here is that we will sooner or later find out that with more data comes more time spent on the same select with a growing relational database. This behaviour is due to numerous reasons, but the main one is that we are bound to make more costly joins with more extensive tables. Our search will be over the same part of the graph with a graph database no matter what happens with the rest. Execution times should therefore be the same.

How to use graph databases with object-oriented languages?

In most applications that communicate with a database, developers use an ORM to create objects from the database. This is a widespread way to use databases, and it is effortless to use. It creates an abstraction between the object-oriented language and SQL. Thus developers do not need a deep knowledge of SQL to use it.

ORMs are powerful, but they are used only with relational databases, which is understandable given that ORM stands for Object-Relational Mapper. If we would like to use a mapper between a graph database and an object-oriented language, we would call it an Object-Graph Mapper (OGM). These types of mappers are not as widespread as ORMs are.

In this thesis, we will go through the steps of creating the OGM library for C#. Here is the outline of this thesis:

- Analyze graph databases
- Analyze Entity Framework and .NET platform
- Analyze existing solutions for OGM library
- Design the library
- Implement a proof-of-concept
- Use Test-Driven Development (TDD) to test the library
- Evaluate the result and propose the next steps

Chapter 1

State of the art of graph databases

The first thing we need to know about graph databases is their definition: "Graph databases store information in graphs, very similarly as relational databases store information in tables and have relations between columns in tables, graph databases store information in nodes and even on the edges, or as we should call them, relations." [2]

To better show the power of graph databases, we will show it in a real-world example. Social networks dominate today's internet content, and everyone wants to be connected with his or her friends and relatives. In one of the social networks, we have people who can post their thoughts or opinions. Other users can follow them to see their posts and like them. We can identify two entities with relations between them. In a relational database, it would look like this.

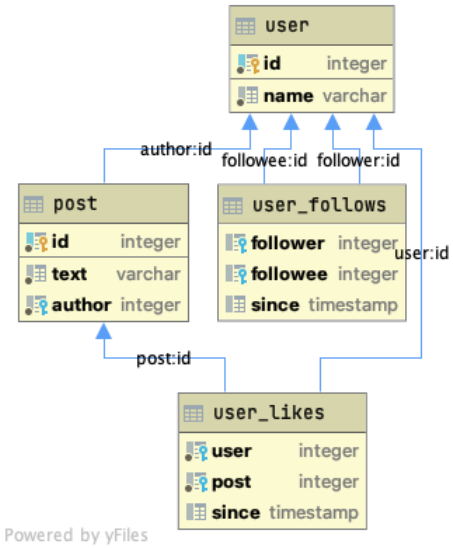


Figure 1.1: Relational database for social network

From the schema in figure 1.1, we can see that we need four tables and five foreign keys to describe all relations between two identified entities. This design can lead to costly joins to answer requests resulting from relationships between entities. For example, we could ask who follows followers who liked someone's post. The query for this example would look like the example in code 1.1.

```
1 SELECT followers_of_followers_data.*
2 FROM post
3 JOIN user_likes fl ON post.id = fl.id_post
4 JOIN "user" followers_liked ON fl.id_follower = followers_liked.id
5 JOIN user_follows followers_of_followers ON followers_liked.id =
  ↪ followers_of_followers.id_following
6 JOIN "user" followers_of_followers_data ON
  ↪ followers_of_followers.id_follower =
  ↪ followers_of_followers_data.id
7 WHERE post.id = :post_id
8 ORDER BY followers_of_followers_data.name;
```

Code 1.1: SQL query for getting followers of users who liked a post

The query in code 1.1 is complex and hard to read. However, the main problem is performance. As data in the database grows, the query execution time will be longer due to the increasing size of the data needed to load. This increase in execution time will happen regardless of the change in the actual result.

Now, we compare the same problem solved using a graph database.

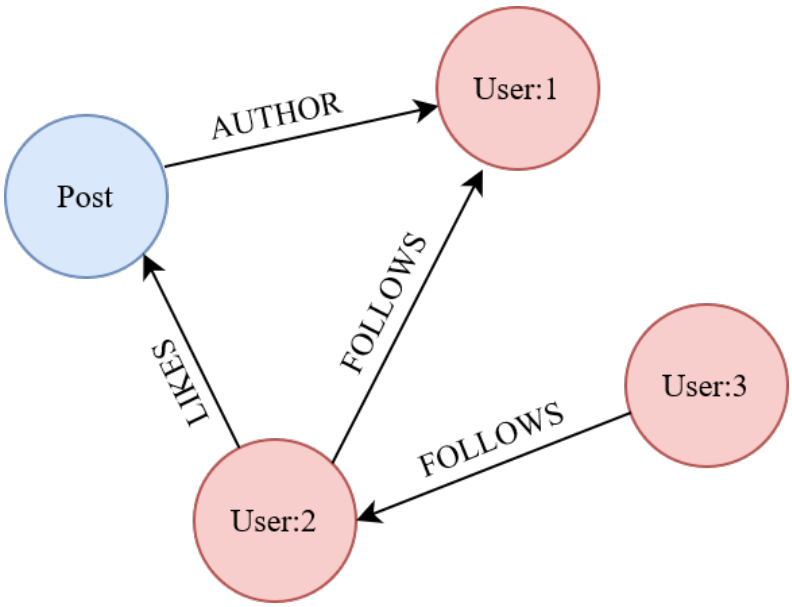


Figure 1.2: Graph database for social network

The figure is the schema from the Neo4j database. Each node has one relation or more to other nodes. Furthermore, the answer to our question from the beginning is already visible. The question was: who follows users that liked the post? To get the answer, we can follow the path in the graph, and as we are going to discover in a moment, the query to get this information also follows this path.

```
1  MATCH (:Post {id:
   ↪  {post_id}})-[:LIKES]->(:User)-[:FOLLOWS]->(followers:User) RETURN
   ↪  (followers);
```

Code 1.2: Cypher query for getting followers of followers who liked a post

As promised, the query in the example 1.2 should not be hard to understand.

This small example was inspired by a blog post from Graham Cox, Introduction to Graph Databases. [3] It is designed to show the strength of graph databases compared to relational databases.

1.1 Native vs Non-native graph databases

When dealing with graph databases, we have to look at two main DBMS features: storage and processing.

Storage is how graphs are stored in memory. If the storage is optimized for graphs, like having related nodes close together, we talk about native graph storage. When implementation uses other NoSQL storage, they are called non-native graph storage.

The second feature is processing, which refers to how graph databases process database operations. What is meant by that is how the database treats queries and how it handles storage. Native databases use Index-free adjacency for processing. [4]

Index-free adjacency: "Native graphs take data that is logically connected via arcs or relationships and hard-wire the physical RAM addresses of these items into the node." [5] This information should answer to why are graph databases faster than other types of DBMS in searching related row, nodes or documents. In traditional RDBMS, looking up a row in another table means pulling an index table representing this relation and then finding a path to the row in said table. This behaviour leads to another problem in RDBMS because the database uses many indexes to keep data connected, which negatively impacts insert operations.

One more thing we should go through is how the graph database handles writes. Connected data requires strict data integrity. Graph databases have to create or update nodes and relationships in one transaction; otherwise, this could result in a corrupted graph, which is almost impossible to fix. The solution to this problem is to write fully ACID-compliant transactions, ensuring that the database will not become corrupted. [4]

1.2 Neo4j

Neo4j is a graph database management system developed by Neo4j, Inc. [6] Neo4j is a native graph database and ACID-compliant. Besides DBMS, the Neo4j company created a custom query language for graph databases called Cypher. [7]

1.3 Cypher

Cypher is a query language created by Neo4j initially for their graph database. Nowadays, it is possible to use Cypher on other graph databases using openCypher, an open-source project for other graph DBMS. [8]

From Neo4j's documentation: Its philosophy is to be easily read and understood by developers, database professionals, and business stakeholders. Its ease of use derives from the fact that it is in accord with how we intuitively describe graphs using diagrams. [9]

1.3.1 Nodes

To depict nodes in Cypher, we surround the node with parentheses, e.g., `(node)`. Parentheses were chosen because they look like circles, a standard visual representation of nodes in the graph. [10]

If we need to refer to the node, we can give it a variable like `(u)` for a user or `(p)` for a post. In real-world queries, full names of variables should be used to understand the query better.

With variables, we mentioned the possibility of giving a variable name to the nodes, but how can we distinguish two nodes from one to the other. We can do this by assigning labels to each node. Labels are like tags or table names, which specify certain entities in the graph. If we look back to our example of users and posts, we already used the two labels: `(p:Post)` and `(u:User)`.

Specifying labels also has another benefit. When not using labels in a query, the database has to look for all nodes, which can negatively impact the performance of the query. [10]

1.3.2 Relationships

Relationships are representations of edges in a graph between nodes. They are marked in Cypher using an arrow `-->` or `<--` between two nodes. Additional information, such as by which relationship type are two nodes connected and

any properties of the relationship, can be placed in square brackets inside the arrow `((p:Post)-[:AUTHOR]->(u:User))`. [10]

The direction of the relationship must be present only while creating the relationship. During the traversal of the graph, it is possible to omit the direction by using two dashes (`--`). This syntax can make queries more flexible and not force users to know in which directions are relationships stored in the database. The tradeoff is a small performance loss.

Like with nodes, variables can be used to refer to relationships. If we do not need to reference the relationship later. We can leave any specification for a relationship using two dashes (`--`).

1.3.3 Nodes and relationship properties

One thing that was not mentioned yet regarding nodes and relationships is properties. Each node and even each relationship can have one or more properties assigned to them.

Properties are name-value pairs providing additional detail. To represent them in the query, we place them in curly brackets. [10] Below is two examples of this usage, one for node and the other for relationship.

- Node property: `(u:User { nickname: 'mr. Incognito' } )`
- Relationship property:
  `-[rel:AUTHOR { posted: 2022-02-05T12:12:20Z } ]`

1.3.4 Querying with Cypher

Cypher has few words reserved for specific actions called keywords like most other programming languages. [11] First, look at the two most common keywords:

- `MATCH`: This keyword is what searches for an existing node, relationship, label, property, or pattern in the database. `MATCH` does work similarly to the `SELECT` in SQL.
- `RETURN`: The `RETURN` keyword defines what values or results we want to retrieve from the database. It is used mainly in search queries as it is not required to be used during writing procedures. `RETURN` does utilize the node and relationship variables. If we want to return any results from

defined `MATCH`, we must specify which nodes, relationships, properties, or patterns we want to return.

1.3.5 Create, update, and delete operations

Besides queries, a proper database system must have methods to create, update or delete data.

The function `CREATE` is used to insert new data into a database. Using `CREATE`, we can create nodes, relationships, and also patterns. Below are some examples (1.3, 1.4) of how `CREATE` is used. [12]

```
1 CREATE (u:User {nickname: "mr. Incognito"})
2 RETURN u
```

Code 1.3: Create a new user with a nickname

In the following example 1.4, we will use `MATCH` to create a relationship between two nodes. If we used the `CREATE` keyword for creating the relationship without the `MATCH`, we would introduce duplicities of both nodes.

```
1 MATCH (u:User {nickname: "mr. Incognito"})
2 MATCH (p:Post {title: "Hello, world!"})
3 CREATE (u)<-[:AUTHOR]-(p)
```

Code 1.4: Create a new relationship between two nodes

There is another way to create this relationship, and we will look at it later in this chapter.

To update data in the database, Cypher uses the `SET` keyword, which can create or update node or relationship properties.

If we want to delete a node or relationship, we use the `DELETE` keyword. This is similar to how SQL `DELETE` works, but with one exception. If a node is in a relationship with another node, we cannot delete it because it would create an inconsistent graph, with a potential relationship pointing to nothing. [12] We

could run two queries to delete the relationship and delete the node itself, but there is a more straightforward solution. We can use `DETACH DELETE`, which does detach all relationships from the node before deleting it.

1.4 ORM

ORM stands for an object-relational mapper based on the object-relational mapping concept. Object-relational mapping is the idea of writing queries using the object-oriented paradigm. There are some limitations to what ORM can accomplish. Developers should always consider these limits before using an ORM framework. [13]

Pros:

- There is no need to use a second language during software development, SQL is a powerful language, but most developers do not use it too often.
- ORM abstracts away from the database system.
- It can lead to better performance than writing queries by ourselves.

Cons:

- If a developer is an SQL power user, he can write queries that will have better performance.
- Developers have to learn how to use ORM properly.
- Developers still need to know how ORM works under the hood.

Using the term ORM with graph databases is not correct. The proper term would be OGM (object-graph mapper), but there are a few reasons why we are using ORM instead of OGM in the name of this thesis. However, please make no mistake. When discussing ORM involving graph databases, it is, in fact, OGM. The main reason is simple: ORM has been around for more than a decade, and developers are familiar with the concept and its challenges.

The differences between ORM and OGM are mainly in the target DBMS. They are not interchangeable, however.

1.5 Summary

This chapter introduced graph databases and compared them to relational DBMS. We compared the queries of both types of databases and their differences. We also studied the differences between native and non-native databases.

The rest of this chapter focused on Neo4j and Cypher language, where we introduced the basics of this language, like how relationships and nodes are defined and base keywords used in Cypher.

In the end, we also adequately introduced the concept of ORM and its relation to OGM.

Chapter 2

State of the art EntityFramework and .NET platform

If anyone wants to create ORM or OGM for the .NET platform in C#, they should go through some principles that are used in EntityFramework and .NET platform. This chapter will go through these principles with insight into the technologies used.

2.1 C#

C# is a general-purpose, type-safe, object-oriented programming language, the goal of which is programmer productivity. To this end, the language balances simplicity, expressiveness, and performance. [14]

Microsoft has created and is developing the C# language. When writing this thesis, the current version of the C# is C# 10.

The C# code is statically compiled down to Common Intermediate Language. CIL cannot be run by itself on a machine. CIL runtime or Common Language Runtime (CLR) must be used. Using Just-In-Time (JIT) compilation, CLR reads CIL and translates CIL to native code or sometimes called machine code. The machine's processor can then read machine code. Using CIL and CLR has benefits in running code cross-platform without recompiling code for different processors, at the cost of some performance. [15]

2.2 .NET

.NET is a framework written for C# and other languages such as F# and Visual Basic, which Microsoft also develops. In their own words: ".NET is an open-source developer platform, created by Microsoft, for building many different types of applications." [16]

The .NET platform went through many changes, and one of them was the introduction of the .NET Core version of the .NET platform. .NET Core is a cross-platform implementation of the .NET platform. At the same time, the .NET Framework was also developed, which was only supported on the Windows platform. The .NET Core and .NET Framework were merged in the .NET version 5.0. From this version on, and the .NET platform has only one version for the whole platform.

To compile a library or program with a .NET platform, developers must first download and install a .NET Software Development Kit (SDK). .NET Software Development Kit (SDK) is either a standalone Command Line Interface (CLI) tool or embedded inside an IDE, for example, in Visual Studio from Microsoft.

One part of the .NET platform we will need for any mapper will be a reflection.

2.2.1 Reflection

The reflection pattern is used to access the class and its methods and fields. We can access the class and its methods and fields without knowing its implementation. This feature has to be supported by the programming language itself. For example, C# supports reflection and is widely used in many popular libraries.

Reflection works by scanning the program's implementations and creating metadata about the classes and methods. This metadata is stored in the program's memory and accessible at runtime.

For example, the following class:

```
public class Person
{
[Key]
public string Name { get; set; }
public int Age { get; set; }
}
```

Code 2.1: Person class with a `Key` attribute

If we would want to know if the class does contain a field with KeyAttribute annotation, we could use the following code 2.2 to get the MemberInfo instance:

```
public bool HasKeyAttribute(Type type)
{
var members = type.GetMembers();
return members.Any(member => member.GetCustomAttributes()
                                   .OfType<EndNodeAttribute>()
                                   .Any());
}
```

Code 2.2: The example of using reflection

We would use this code in cases where we do not know how an object is implemented.

2.2.2 LINQ

In the previous section, we used a method called `Any` to check if a collection contains an element. This method is part of a library in .NET called LINQ. LINQ stands for Language Integrated Query, and it is a library that provides a set of methods that can be used to query objects. We can filter, order, group, and transform data using this library.

LINQ is internally working as an expression tree, each command as an expression. The expression tree is immutable and evaluated at runtime, and developers can use the tree to analyze and convert it to SQL, for exam-

ple. To extend expression tree capabilities, LINQ provides an abstract class `ExpressionVisitor`, called for each expression combined with extension for `IQueryable<T>` and other tools.

With LINQ, we can write queries in a more readable way. These queries can then be translated to SQL queries, for example. This feature is used in Entity Framework.

2.3 Entity Framework Core

The Entity Framework is an ORM with the support of writing queries using LINQ expressions.

When we are talking about Entity Framework, we are talking about the latest version of this framework, Entity Framework Core or EFCore, as it is known in the community. This version was released for .NET Core 1.0, the first Microsoft version of .NET purposely built for multiplatform use.

To use Entity Framework, developers must add another dependency to their projects. This dependency is for Entity Framework and its called provider, which is used to provide connection and extend translation capabilities for Entity Framework to work correctly over a specific database.

If we want to know how Entity Framework translates LINQ queries to SQL queries, we need to look at the implementation of the Entity Framework. The translation is separated into several parts, but the most important parts are implementations of the `QueryableMethodTranslatingExpressionVisitor` class which translates the LINQ method into custom expression, and the `QuerySqlGenerator` which translates the expression into SQL. There are more expression visitors and generators, but these are the most important ones.

2.3.1 `DbSet`

An inseparable part of the Entity Framework Core is the `DbSet`. This class is used to query data from the database. The Entity Framework creates it, and it is an implementation of `IQueryable<T>` interface. This interface is the backbone of LINQ.

With this abstract class, we can create queries to the database and add or update entities to the change tracker. Change tracker is then used for saving operation, where it is checked what changes in objects were made and then

they are saved to the database using `DbContext.SaveChanges` method or its asynchronous version `DbContext.SaveChangesAsync`.

`DbSet` implements methods in both synchronous and asynchronous versions. The main benefit of asynchronous versions is that they are not blocking the thread. They are using `Task` to run the code on another thread from a thread pool or outside of the application's thread pool if possible. This is done by using `async` keyword.

To show the exact process of generating a query from LINQ to SQL, we will go through the implementation of one of the methods in the `DbSet` class.

2.3.2 `FindAsync`

The `FindAsync` method is used to find an entity by its primary key. If the entity is not found, it returns `default` value for given object. It takes 1 to n parameters of type `object`. These parameters are mapped to the primary keys of the entity. If the number of parameters is not equal to the number of primary keys, the exception is thrown.

We are now going to analyze the implementation of the `FindAsync` method. The code for this method and the whole Entity Framework is available in the GitHub repository at `https://github.com/dotnet/efcore`.

Inside the concrete implementation of the `DbSet<T>` abstract class the `FindAsync` method call method from `IEntityFinder` inteface called `FindAsync`. The `IEntityFinder.FindAsync` method is what starts the translation process of the query by creating the source LINQ query. The find method is used the `FirstOrDefaultAsync` method, which is equivalent to `FirstOrDefault` method in LINQ. This method accepts a lambda function as a predicate for the query. This predicate is build inside the `EntityFinder` class.

The `FirstOrDefaultAsync` is an extension method for `IQueryable` interface, and as the name suggests, it is used for retrieving the first element of a sequence or a default value if the sequence contains no elements. This extension method starts the process of translation and enumeration of the result from the database.

Translation of the query works by overriding the default provider for `IQueryable` done insede the `DbSet<T>` abstract class. The custom provider is defined by the `IAsyncQueryProvider` interface. If we skip to the translation

process itself, we will find ourselves in the `QueryCompilationCOntext` class inside, which is a method `CreateQueryExecutor`. This method is responsible for transforming the expression tree for our query. The process is divided into a few steps preprocessing, translating methods, and postprocessing. At the end of this method is the translation of the expression tree into a custom lambda function, which is used to execute the query, enumerate the result and return it. Each step of the translation process uses an implementation of the `ExpressionVisitor` abstract class with different overrides.

2.4 Summary

We are now acquainted with Entity Framework and some of its principles. This chapter gave us an idea of what it means to implement OGM with LINQ support. Based on this research, we will build the part of the library responsible for translating LINQ to Cypher.

Chapter 3

Existing OGM libraries for graph databases

Searching for OGM for Neo4j in C# did not result in any working, well maintained, and well documented OGM library. However, there is a library written by the Neo4j company itself in Java called Neo4j–OGM. In this chapter, we will study this library and conclude its relevance for our goal of building a OGM library for the .NET platform.

3.1 Neo4j–OGM

Neo4j company has already created an OGM for their DBMS. It supports dynamic objects and maps nodes and their relations into the domain model written in Java.

The list of features from the official documentation: [7]

- Object graph mapping of annotated node- and relationship-entities
- Neo4jSession for direct interaction with Neo4j
- Fast class metadata scanning
- Optimized management of data loading and change tracking for minimal data transfers
- Multiple transports: binary (bolt), HTTP and embedded
- Persistence lifecycle events

- Query result projection to data transfer objects (DTO)

The following sections are information obtained from the official documentation of Neo4j–OGM for Java. [17]

3.1.1 Neo4j drivers

There are three possible drivers to use, Bolt driver, HTTP driver, or embedded driver, which creates an in-memory Neo4j database instance.

Using a different driver in the development or test environment will not affect the production code. These drivers are interchangeable without the need for modification in queries.

3.1.2 Entities

The library offers the possibility to define and shape entities and relationships. `@NodeEntity` annotation is used to declare that a Plain Old Java Object (POJO)is a representation of a node. This class must have one empty public constructor to allow the library to construct the objects.

Fields on the entity are by default mapped to properties of the node. Fields referencing other node entities (or collections) are linked with relationships.

If we want to change fields name or other properties, we can use annotations like `@Property`, `@Id`, `@GeneratedValue`, or `@Relationship`. On the other hand, if we want to not include a field in the node, we can use `@Transient` annotation.

3.1.3 Relationships

Every field of an entity that references one or more other node entities is backed by relationships in the graph. These relationships are managed by Neo4j–OGM automatically.

If we want to specify relationship properties, like the direction of the relationship, the `@Relationship` annotation is used. The directions are either `INCOMING`, `OUTGOING`, or `UNDIRECTED`, where the last one ensures that the path between two node entities is navigable from either side.

Relationships in a graph database can have properties assigned to them. Neo4j–OGM supports this feature using an (POJO) with annotation `@RelationshipEntity`.

A String attribute called “type” is available on the `@RelationshipEntity` annotation to control the relationship type. Like the simple strategy for labelling node entities, if “type” is not provided, the class’s name derives from the relationship type, although it is converted into a snake case with an upper casing to honour the naming conventions Neo4j relationships. [17]

Inside the entity, we then define `@StartNode` and `@EndNode` annotations. In referenced entities, we also define a reference to the related entity and use `@Relationship` annotation with the same type as is in `@RelationshipEntity` annotation.

In the code example 3.1 is a simple example of using `@RelationshipEntity`.

```
@NodeEntity
public class Actor {
 Long id;
 @Relationship(type="PLAYED_IN") private Role playedIn;
}

@RelationshipEntity(type = "PLAYED_IN")
public class Role {
 @Id @GeneratedValue private Long relationshipId;
 @Property private String title;
 @StartNode private Actor actor;
 @EndNode private Movie movie;
}

@NodeEntity
public class Movie {
 private Long id;
 private String title;
}
```

Code 3.1: An example of model with relationship entity

3.1.4 Indexes

Indexes are also defined by using an annotation. We already saw one of them: the `@Id` annotation used for the primary index.

Primary indexes are not the only type of index we can define in our model.

We can also define indexes for other properties using `@Index` annotation, and the index will have unique constraint if we use `@Index(unique=true)`.

This library also supports composite indexes and node constraints with `@CompositeIndex` and `@CompositeIndex(unique = true)` annotations, respectively.

`@Required` is an existence constraint. "It is possible to annotate properties in both node and relationship entities. For node entities the label of declaring class is used to create the constraint. For relationship entities the relationship type is used — such type must be defined on leaf class." [17]

The library can handle creating and managing indexes or constraints, but as stated in the documentation [17], this feature should be used only for development and not in production. That is why this feature is, by default, turned off.

These are the available modes for managing indexes and constraints:

node	Default, nothing is done on the side of the OGM library.
validate	This ensures that all constraints and indexes are in the database before starting up.
assert	This drops all indexes on startup and then creates only these defined by OGM annotations.
update	Update indexes and constraints based on annotations.
dump	Dumps all indexes and constraints to a file.

Table 3.1: Available modes for indexes and constraints

3.1.5 Sessions

To interact with mapped entities, Neo4j–OGM requires an instance of the `Session` class, which can be created by using `SessionFactory` class. Besides creating `Session`, `SessionFactory` also setups up the object graph mapping metadata when constructed. The metadata are used across all `Session` instances created by `SessionFactory`.

Session keeps track of mapped entities, their changes, and changes in their relationships. Tracking is then used when saving or otherwise working with mapped entities. When an entity is loaded by an instance of the `Session` class, the result is cached within the `Session` instance.

To keep new data and not prolong sessions too much, session lifetime can be managed in code. Too long session lifetime means that other users can change data causing concurrency exceptions, and too short a lifetime means costly save operations will be executed more often. There is a way to force the session's cache to clear, but it is advised against it.

Neo4j–OGM use Cypher queries only for its operations, which limits the capabilities of the Neo4j–OGM library. Documentation suggests using server-side operations for more complex or performant graph traversals over the graph. Nevertheless, Cypher should be powerful enough for most of the problems.

3.1.6 Persisting entities

The session allows to save, load and delete entities with transaction handling and exception translation managed. Persistence is performed through method save. This method then looks at underlying `MappingContext` and compares data loaded from the database with the saved entity, creating appropriate Cypher queries to update the database based on differences. Calling save is necessary to propagate changes because Neo4j–OGM does not automatically commit changes.

The save method has a second optional parameter: the depth, which can restrict how deep will the save operation in the graph for given entities. The default value is `-1`, which means saving every change in node and all reachable nodes from it into the database. This approach is recommended because of possible inconsistencies that could happen.

3.1.7 Loading entities

Loading entities can be done using methods `session.loadxxx` or writing a custom Cypher query with methods `session.query` and `session.queryForObject`. Like the depth parameter for the saving function, the load functions also have a depth parameter.

Depth is there to determine how many depths of relatives will be loaded with the query. The default behaviour is to load the object's properties and neighbours. This behaviour represents loading data using a depth set to value 1. Depth is mainly helpful when loading deeper than broader parts of a graph and helps developers execute fewer load operations from the database.

When using load methods from the session, the session uses `LoadStrategy`

to generate a `RETURN` clause. The default strategy is schema loading, which uses entities metadata. The other is the path load strategy that uses paths from the root node. It is possible to change the strategy for a query using `Session.setStrategy` or globally by calling `SessionFactory.setStrategy`.

3.1.8 Transactions

Neo4j uses transactions, which means queries can be executed only in transaction boundaries. Neo4j–OGM offers tools to manage transactions. The developer does not have to use them because the session handles them independently. However, with the auto-commit transaction.

3.2 Summary

This chapter introduced the Neo4j–OGM library, its features, and how to use them. We studied them to understand what is needed for an implementation of an OGM in any language.

Chapter 4

Design of OGM library

We should now have all information we need to propose our solution for OGM using the C# language and .NET platform. Before we start designing individual pieces of the library, we need to define a list of requirements that should the library meet.

The solution should be able to:

- connect to a database
- map objects into graph structure
- map LINQ query into Cypher query
- execute a command in a database
- retrieve a result from the database
- map the result from the database into objects

With these minimal requirements set, we can now go through them, analyze them, and design individual solutions for them.

4.1 Connect to a database

Neo4j company has created a client for .NET that supports both "bolt" and "neo4j" URI schemes. [18] This driver is a NuGet package, publicly accessible and licensed under Apache 2.0 license. This means we can use this driver in our library as a dependency.

The library should handle the driver's lifecycle. During startup, the application should create an instance of the driver and then correctly destroy this instance on exit.

In the figure 4.1 is a visualisation of connections between components.

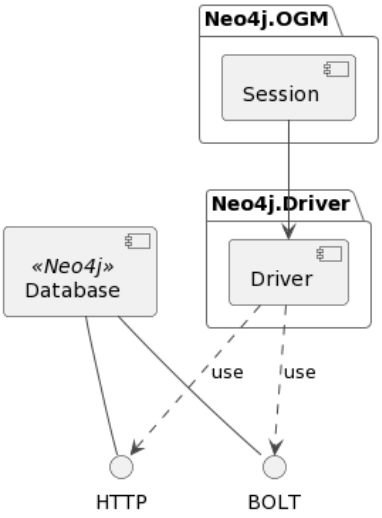


Figure 4.1: Components diagram

We will use Neo4j's official driver for connection, but we need to encapsulate it into our library. We will define a set of parameters for our library to ensure we have everything we need to create a connection to the database.

4.2 Map objects into graph structure

To create Cypher queries from the objects, we need to know the graph structure described in a user's domain.

What we need to do is build metadata. To build metadata, we first need information, which assemblies contain models representing nodes and relationships in a graph. The end-user of our library must declare these assemblies as it would be slow for our library to scan all available assemblies.

4.2.1 Annotations

To create a metadata object, we need to identify and process nodes and relationships. We need to have a way for end-users to describe each node and its relationship with all properties they want to define.

We already have a solution to this problem, and Neo4j–OGM uses it too to solve the same issue. We will use annotations using attributes to describe nodes and their relationships.

We will look for these annotations during initialization using reflection, which is well supported by C# and the .NET platform. With annotation, we can describe the graph and create the metadata object.

4.2.2 Entity mapper

Entity mapper is a part of our library responsible for mapping entities into builders, which can be translated into Cypher queries.

For this purpose, we will define a interface `IEntityMapper`. This interface will contain this list of public methods:

- `Map`: this method map an entity to a `ICompilerContext`
- `CompilerContext`: this method returns a current instance of `ICompilerContext`

In the picture 4.2 is class diagram showing entity mapper part of the library. The interface `ICompilerContext` contains methods for controlling the context of mapped nodes and relationships. These methods are used during mapping an entity in the `IEntityMapper.Map` method.

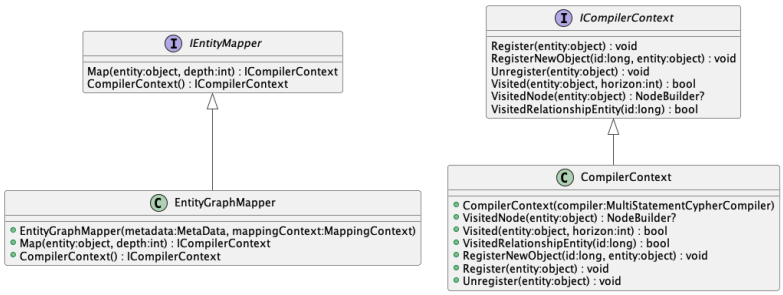


Figure 4.2: `IEntityMapper` and `ICompilerContext` class diagrams

4.3 Map LINQ query into Cypher query

Mapping a LINQ query to a Cypher query is a bit more complicated. As we already know, from our analysis of the Entity Framework, LINQ is translated into a custom expression tree and then into a lambda function which executes the query. We will have to create a custom expression tree similar to the one

used in the Entity Framework but optimized for Cypher language. We can illustrate the process of transformation using the state diagram 4.3.

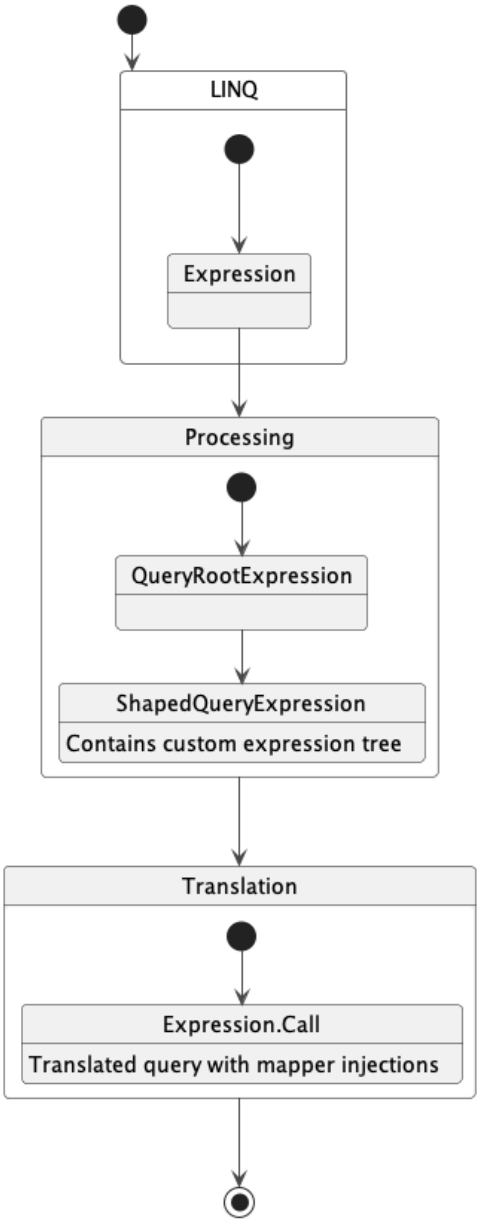


Figure 4.3: LINQ expression transformation

4.3.1 `IQueryable` extension

If we want to communicate with the database, the `IQueryable` instance must have a correct data provider. This provider must be able to transform the ex-

pression tree into a Cypher query and enumerate the result from the database asynchronously because of the limitation of the driver. We are going to extend `IQueryProvider` interface with `IAsyncQueryProvider` interface which declares `IAsyncQueryProvider.ExecuteAsync<T>` method.

To set a correct provider we are going to define new class that implements `IQueryable<T>` interface called `DbSet<T>`. The name of the class is the same as it is in Entity Framework. Besides implementing `IQueryable<T>` interface `DbSet<T>` class also implements `IAsyncEnumerable<T>` interface, because we will use asynchronous enumeration.

Because we are communicating with a database using asynchronous operations, we need to create extension methods for `DbSet<T>` which are asynchronous. For example, LINQ has method called `FirstOrDefault` which returns first element of `IQueryable<T>` or default value of type T. We will ahve to create an extension of `IQueryable<T>` with method `FirstOrDefaultAsync`. This will be similar to most of the methods from LINQ.

Inside this extension method, we will call providers `IAsyncQueryProvider.ExecuteAsync<T>` method. This is our entry point, from which we will start a translation of the expression tree into a Cypher query and enumerate the result.

For better visualisation, here 4.4 is class diagram of query provider and `DbSet<T>` class.

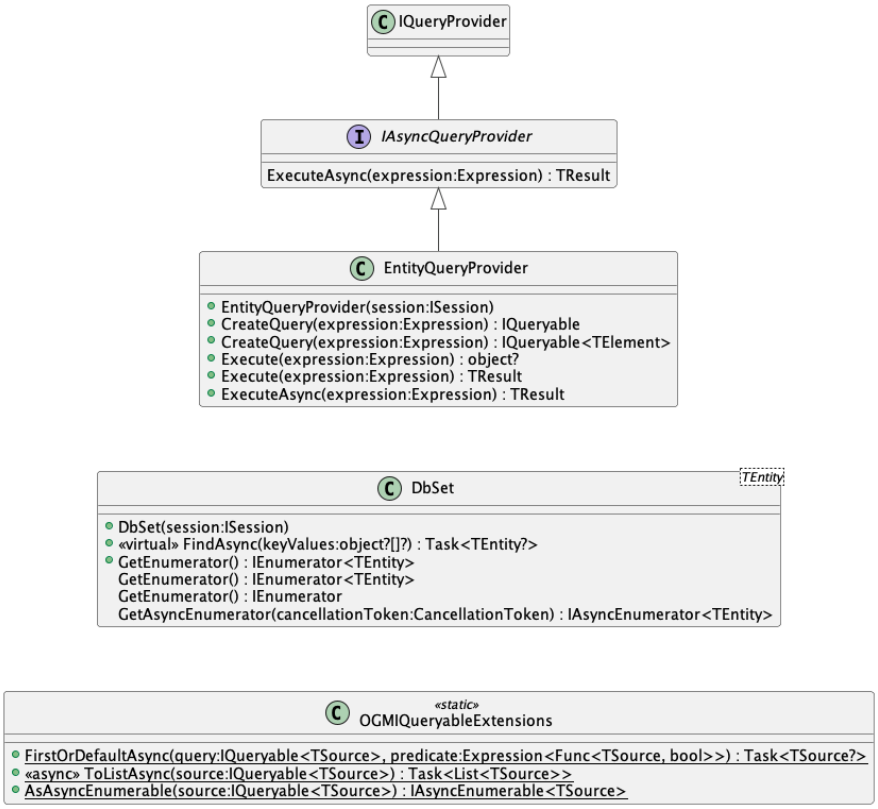


Figure 4.4: `IAsyncQueryProvider` and `DbSet<TEnity>` with extension class diagram

4.3.2 Query compilation

To best visualize the process of query compilation, we will use sequence diagram 4.5. In this diagram, we are using different implementations of the `ExpressionVisitor` abstract class to be used in the different steps of the translation of a LINQ query into a Cypher query.

The result of this compilation will be a `Func<QueryContext, TResult>`, which is an object representing the function with the instance of the `QueryContext` class as a paramater and return object of `TResult` type. This function does both a query execution and enumerates a result from a database response.

In the sequence diagram 4.5, are three expression tree visitors, each serving different purpose. Using these visitors gives us the ability to quickly expand our library's capabilities. We can also further expand the capabilities of defined visitors inside their implementation.

We should also introduce the concerns of each visitor. The first one that is used in the diagram is `ParameterExtractingExpressionVisitor`, which extracts parameters from the original expression.

The next one is `QueryableMethodTranslationExpressionVisitor`, this visitor will be responsible for translating the original LINQ query into an expression tree that will be consisted of `CypherExpression` derivatives, which will copy the Cypher language structure.

The last visitor is `ShapedQueryCompilinExpressionVisitor`, it is responsible for taking a `ShapedQueryExpression` and translating it into a `Expression.Call` expression, which can be then compiled into the `Func<QueryContext, TResult>` object.

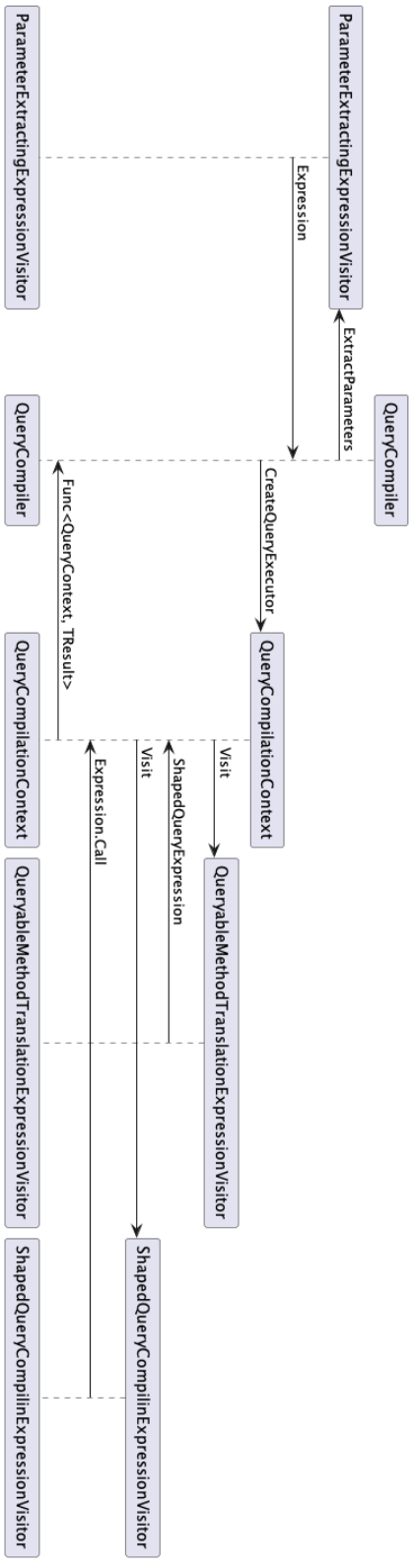


Figure 4.5: `QueryCompiler` compilation sequence

4.4 Execute a command and retrieve the result

We have already decided to use the official Neo4j .NET driver for communications with the database. From this library, the result of any query is returned using the `IResultCursor` interface. This interface is somewhat similar to an asynchronous enumerator, but it does not implement the `IAsyncEnumerable<T>` interface. The `IResultCursor` interface declares methods as show on the 4.6 picture. We will have to adapt our code to use this interface. We are going to use these methods inside a proper implementation of `IAsyncEnumerable<T>` interface.

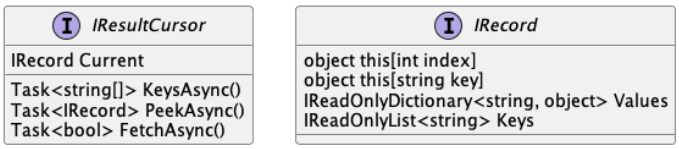


Figure 4.6: `IResultCursor` and `IRecord` interfaces

4.5 Map the result of the query to an object

Mapping the result into an object is our last step. We already know from 4.6 class diagram, that we can access an `IRecord` interface using `IResultCursor.Current` property. This interface represents a single result of the query, which is defined in Cypher using the `RETURN` clause.

Our mapper needs to read the result and correctly choose the right key from the values. `IRecord` is not a representation of a single node or relationship, but it is a representation of the `RETURN` clause, meaning that it contains all the values of the `RETURN` clause. Mapper needs to know which key contains which entity and or value.

We can solve this issue by creating an extension method that will extend an `IRecord` interface and accept a `string` parameter defining an alias of value. This method will return either a value or an entity.

4.6 Summary

In this chapter, we designed a solution for an OGM library in .NET with LINQ to Cypher translation. We started with defining six critical requirements that our library must solve and then went through each one and proposed a solution for them.

We defined how we would handle creating a connection to the database using Neo4j's official driver for the .NET platform. Then we moved on to the problem of mapping objects into graph structure using annotations and reflection. We also proposed a solution for mapping LINQ queries to Cypher queries. At the end of the chapter, we went over the process of executions and mapping the Cypher query and its result into objects.

With this design, we should have all that we need to successfully implement OGM library for .NET with LINQ to Cypher translation.

Chapter 5

Implementation of proof-of-concept

With the library's design done, we can now start implementing proof-of-concept. In the beginning, we should define the goals we want to accomplish in the proof-of-concept. These are the goals:

- Create or update nodes in a database
- Map LINQ to Cypher

Before we start the implementation, we also need to introduce the tools used. The main tools used are Visual Studio Code with multiple plugins like Copilot and C# extension. Github is used as VCS. Code is available at this URL `https://github.com/TomStary/dotnet-neo4j-ogm` or in the appendix of this thesis.

To create a new project in .NET, we will use the following command in the terminal application: `dotnet new classlib` with a parameter for the name of the project. We also want to separate tests from the source code of the library, so we employ a file structure like this 5.1.

```
dotnet-neo4j-ogm
├── src
│   └── Neo4j.OGM
├── tests
│   └── Neo4j.OGM.Tests
├── .gitignore
└── Neo4j.OGM.sln
```

Figure 5.1: File structure

With the file structure prepared, we can begin our implementation. During the development of this library, we will use Test-Driven Development (TDD). We will not go deep into this approach in this chapter, as it will be described in the next chapter, but keep in mind that during development, TDD was used as it is an excellent way to develop libraries.

5.1 Common infrastructure

If we want to achieve set goals for proof-of-concept, we will need a common infrastructure like the implementation of `ISession` interface. This interface can be described as an entry point for all of our operations with the database. It will handle both saving entities and creating `DbSet<T>` instances, which can be used to query over entities using LINQ.

For the session to work properly, we need to have a few things:

- connection to the database
- domain metadata

The connection to a database and domain metadata can be acquired from the session's constructor. However, the best solution is to use `SessionFactory` instead. Using a factory has many benefits. The client's code will not be responsible for creating a connection to the database for each instance of the `Session` class. It will also have a cached domain metadata object. Another benefit is that it will be possible to hide the concrete implementation of `ISession` interface.

Implementation of `SessionFactory` is simple, we create a class with constructor accepts three parameters:

- `string connectionString` — contains connection string to the database

- `IAuthToken token` — token created by `AuthTokens` class from Neo4j drivers library, used to authenticate connections to the database
- `params Assembly[] assemblies` — assemblies containing domain models

Using the first two parameters, we can create `IDriver` instance, which will be used to create session instances. The third parameter is special because the type of the parameter is prefixed with keyword `params`. It means we can call the constructor with as many instances of `Assembly` as we want (we are limited only by language itself, which has a cap at 2^{14} parameters). From C# documentation: "No additional parameters are permitted after the `params` keyword in a method declaration, and only one params keyword is permitted in a method declaration." [19]

The assemblies refer to where domain models are located; we need from the client's code information which assemblies to scan using reflection to pick up classes representing nodes and relationships. We will use this information during building metadata objects.

5.1.1 Building metadata

We described the importance of metadata, but how are we going to obtain them. We already have assemblies from client's code, that should contain the domain model. We use these assemblies in `MetaData` class constructor. Here is an actual implementation of `MetaData` constructor:

```
1  /// <summary>
2  /// MetaData constructor.
3  /// </summary>
4  /// <param name="assemblies">Assemblies containing domain
   ↪  model.</param>
5  public MetaData(params Assembly[] assemblies)
6  {
7   _domainInfo = new DomainInfo(assemblies);
8   Schema = new SchemaBuilder(_domainInfo).Build();
9  }
```

Code 5.1: `MetaData` constructor

We can now see how are the assemblies passed to the `DomainInfo` construc-

tor. Inside this class, we are going through all the assemblies and scanning all classes obtained by the `Assembly.GetType` method. This method returns an array of `Type` objects representing all the types in the assembly. We then check each `Type` if it has an annotation for the node or relationship. Using our custom extension methods for the `Type` type, `HasNodeAttribute` and `HasRelationshipEntityAttribute`. If the type has the annotation, we will add it to the correct dictionary, which will be used while building the schema. We can see how this is done in the code example 5.2.

```
1  /// <summary>
2  /// Check if given <see cref="Type" does have the <see
   ↪  cref="NodeAttribute"> applied as custom attribute.
3  /// </summary>
4  internal static bool HasNodeAttribute(this Type type)
5   => type.GetCustomAttributes().Any(attribute => attribute is
   ↪   NodeAttribute);
6
7  /// <summary>
8  /// Check if given <see cref="Type" does have the <see
   ↪  cref="RelationshipEntityAttribute"> applied as custom attribute.
9  /// </summary>
10 internal static bool HasRelationshipEntityAttribute(this Type type)
11  => type.GetCustomAttributes().Any(attribute => attribute is
   ↪   RelationshipEntityAttribute);
```

Code 5.2: `HasNodeAttribute` and `HasRelationshipEntityAttribute` extension methods

5.1.2 The `internal` keyword

Keen reader might caught it up, but in the last code example 5.2 we used the keyword `internal` as an access modifier to the defined methods. This access modifier makes methods, classes, and properties available only inside the assembly. More on this subject can be read in the official documentation of C# language [20].

5.1.3 Building schema

Building schema is the last step in creating metadata for the client's domain model. We use `SchemaBuilder` class to build the schema. We need to pass the `DomainInfo` instance to the `SchemaBuilder` constructor to use information ob-

tained from scanning assemblies. At the start we create new `Schema` instance, which is our concrete implementation of `ISchema` interface.

`SchemaBuilder.Build` method is responsible for building the schema. It does that by iterating over nodes and relationships from the instance of `DomainInfo` class and adding them to `Schema` instance using `ISchema.AddNode` and `ISchema.AddRelationship` methods. The resulting schema is then returned.

We now have both schema and driver for creating the session. We will store both of them inside `SessionFactory` instance and use them every time a new instance of `Session` is requested. To help client's code to manage instances of `SessionFactory` we will create an extension of `IServiceCollection` which is a component of .NET responsible for managing the Dependency Injection Container (DIC). The lifetime of the `SessionFactory` is a singleton, meaning that only one instance will be created during the runtime of the application. The extension implementation is shown in the code example 5.3.

```
1  /// <summary>
2  /// Try and register the Neo4j OGM services in the DI container.
3  /// </summary>
4  public static IServiceCollection AddNeo4jOGMFactory(
5   this IServiceCollection serviceCollection,
6   string connectionString,
7   IAuthToken authToken,
8   params Assembly[] assemblies
9   )
10 {
11  serviceCollection.TryAddSingleton(
12  new SessionFactory(connectionString, authToken, assemblies));
13  return serviceCollection;
14 }
```

Code 5.3: `IServiceCollection` extension method

With this extension method done, we have finished the common infrastructure, and we can now go and start implementing our goals from the beginning of this chapter.

5.1.4 `DbSet<T>`

`DbSet<T>` is a generic class that represents a set of entities of a given type. This concept is very similar to the `DbSet<T>` abstract class from the Entity Framework library.

Inside the constructor of `DbSet<T>` class, we are going to set the correct provider as well as set an instance of `ISession`. Because we need an instance of `ISession` we are going to declare a method inside this interface called `ISession.Set<TEntity>` which will create a new instance of `DbSet<T>`.

5.2 Create or update nodes in a database

In the code example 5.4 is an actual implementation of the save operation inside `ISession` concrete implementation. We are going to use the `EntityGraphMapper` class to map the entity/entities into a structure that will be translated into `IStatement` objects which represent statements that will be executed in the database. During mapping, we will also note the relationships between nodes using information from `MetaData` object we built for this purpose. This whole concept is borrowed from the official implementation of the Neo4j–OGM library for Java.

```
1  /// <summary>
2  /// Save entity or list of entities to the database.
3  /// </summary>
4  public async Task SaveAsync<TEntity>(TEntity entity) where TEntity :
   ↪  class
5  {
6   CheckDisposed();
7
8   // tranform entity/entities into array
9   IEnumerable<TEntity> objects;
10  if (typeof(IEnumerable).IsAssignableFrom(entity.GetType()))
11  {
12  objects = (IEnumerable<TEntity>)entity;
13  }
14  else
15  {
16  objects = new[] { entity };
17  }
18
19  // map objects into a graph and creates statements
20  foreach (var item in objects)
21  {
22  _entityGraphMapper.Map(item, -1);
23  }
24
25  // execute statements
26  await ExecuteSave(_entityGraphMapper.CompilerContext());
27 }
```

Code 5.4: Internal implementation of save operation

5.2.1 `EntityGraphMapper`

The `EntityGraphMapper` is responsible for mapping entities to builders that will be translated into Cypher, as was stated in the paragraph before. The resulting structure of this translation is saved inside a compiler context. The compiler context is a container for all the information used during mapping and generating Cypher queries. It tracks visited nodes as well as visited relationships.

Inside the `ICompilerContext` is saved the instance of compiler, in this case it is

`IMultiStatementCypherCompiler`, which is responsible for generating create and update Cypher queries. This compiler is used inside the `Session.ExecuteSave` method.

5.2.2 IMultiStatementCypherCompiler

```
1  public interface IMultiStatementCypherCompiler
2  {
3   CompilerContext Context { get; }
4   NodeBuilder CreateNode(long id);
5   IEnumerable<IStatement> CreateNodesStatements();
6   NodeBuilder ExistingNode(long id);
7   RelationshipBuilder ExistingRelationship(long relId, string type);
8   IEnumerable<IStatement> GetAllStatements();
9   bool HasStatementDependentOnNewNode();
10  RelationshipBuilder NewRelationship(string type);
11  RelationshipBuilder NewRelationship(string type, bool
    ↪  mapBothDirections);
12  void UseStatementFactory(IStatementFactory statementFactory);
13 }
```

Code 5.5: `IMultiStatementCypherCompiler` interface

This interface defines a set of methods needed for preparing and generating `IStatement` instances that will then be executed on the database. The `NodeBuilder` and `RelationshipBuilder` classes are created by `EntityGraphMapper` and are the building blocks for `IStatement` instances.

5.2.3 IStatement

The `IStatement` was mentioned multiple times. It is a simple wrapper around a string containing the Cypher query and dictionary of parameters used in the query. These two values can then be used to generate `Query` object from Neo4j.Driver library. `Query` is used by Neo4j.Driver in their transaction implementation.

5.3 Mapping LINQ to Cypher

The process of mapping LINQ to Cypher can be divided into these steps:

- prepare LINQ query
- map LINQ to our expression tree
- map our expression tree into Cypher
- map result from database into result object.

We will look at each step and show some parts of the implementation.

5.3.1 Prepare LINQ query

Before we translate the query, we have to prepare our LINQ query. Preparation starts with the definition of the asynchronous method in the `DbSet<T>` class or the `IQueryable<T>` extension of an asynchronous method. The best example of this definition would be method `FindAsync` as it is defined in the `DbSet<T>` class, but it also uses an extension method for `IQueryable<T>` interface. The reason behind using asynchronous methods is the implementation of `IDriver` which has only asynchronous methods for working with transactions.

Apart from the checks of key values, the method will also create a lambda expression that will be translated into the `WHERE` statement in the Cypher query. This lambda expression is then passed to the `FirstOrDefaultAsync` method, which is an extension of the `IQueryable<T>` interface as we discussed above. The code example 5.6 shows the exact implementation of the `DbSet<T>.FindAsync` method. Inside the extension method of `FirstOrDefaultAsync`, we are calling the `IAsyncQueryProvider.ExecuteAsync<T>` method. This method is responsible for the translation and execution of the query. The translation is driven by our implementation of `ExpressionVisitor` abstract class.

```
1  public virtual Task<TEntity?> FindAsync(params object?[]? keyValues)
2  {
3   if (keyValues == null
4   || keyValues.Any(key => key == null))
5   {
6   return Task.FromResult<TEntity?>(default);
7   }
8
9   var keyProperties = typeof(TEntity).GetProperties()
10  .Where(property => property.GetCustomAttribute<KeyAttribute>() !=
    ↪  null)
11  .ToArray();
12
13  if (keyProperties.Length != keyValues.Length)
14  {
15  throw new ArgumentException("Incorrect number of key values");
16  }
17
18  return this.FirstOrDefaultAsync(BuildLambda(keyProperties, new
    ↪  ValueBuffer(keyValues)));
19 }
```

Code 5.6: `DbSet<T>.FindAsync` implementation

5.3.2 Expression visitors

Our goal is to create a custom expression tree representing the Cypher query translated from the LINQ source.

First of all, we will introduce classes to which we want to translate our LINQ expression tree 5.2.

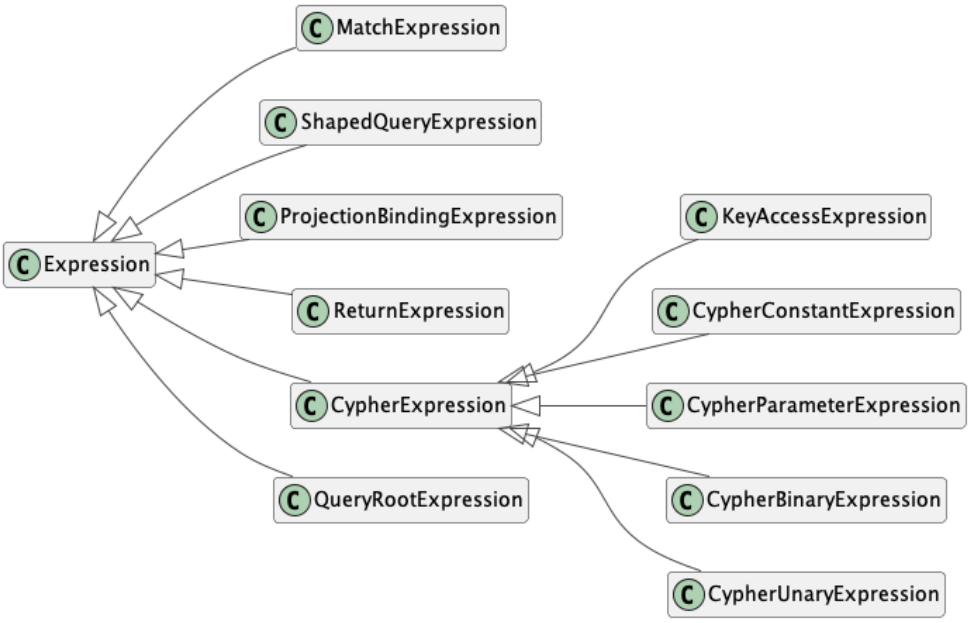


Figure 5.2: Custom classes for translating LINQ expression tree

To translate LINQ query to our custom expression tree, we will use the `QueryableMethodTranslationExpressionVisitor` class. This class implements the `ExpressionVisitor` abstract class from LINQ. Our query is starts as `QueryRootExpression` which is translated into `ShapedQueryExpression`. The `ShapedQueryExpression` contains the `MatchExpression` and `EntityShaperExpression` expression. The `MatchExpression` represents the `MATCH` statement in the Cypher query and `EntityShaperExpression` is used for mapping the result of the query from the database. This translation is done inside `VisitExtension` method which is called for each expression marked with `ExpressionType.Extension` as their `NodeType` property.

To set up `MatchExpression`, we are using a method `VisitMethodCall` which visits the children of the `MethodCallExpression`. This method translates calls like `FirstOrDefault` from LINQ library. Inside this method the `WHERE` clause is translated as well as `MatchExpression` other properties like `MatchPattern.Limit`. Also, the `ReturnExpression` is created inside this method, which is equivalent to the `LIMIT` keyword inside the Cypher query.

The result of `Visit` method is a `ShapedQueryExpression` which contains the

`MatchExpression` with `ReturnExpression` and other predicates and expressions. The `ShapedQueryExpression` is now ready to be translated into Cypher query.

The visitor responsible for the actual translation to Cypher is `CypherExpressionVisitor`. Inside this implementation of the `ExpressionVisitor` abstarct class, we declare methods that corresponds with our custom classes, like `VisitCypherBinary` or `VisitMatch`. Each of these methods translates part of the expression tree into a Cypher query or calls further visits using the `CypherExpressionVisitor` class. When the translation is done, the generated query is used in our custom enumerator, enumerating the result from the database. This enumerator is implemented inside `QueryingEnumerable<T>` class.

With the enumerator done, and the query generated, we need to build a lambda function which will return the result. To do this, we will use another visitor. In this case, this visitor needs to visit our `ShapedQueryExpression` and create a right call that enumerates and returns the result. The name of the class is `ShapedQueryCompilingExpressionVisitor` and it sets up the lambda function with use of the `ProjectionBindingExpression` to translate the `IRecord` result from the drivers library into our result object.

5.4 Summary

This chapter was about implementing the library we designed in the chapter before. We described the structure of our project and then implemented two critical goals of our proof-of-concept. With the first goal, we implemented the save operation for entities. We used metadata with information about graph schema to properly create nodes and relationships. The second goal was mapping LINQ to Cypher query and extracting results from the database structure; we had to implement our own expressions structure to transform the expression tree so that we would be able to translate it to Cypher query. We accomplished both of our goals and are ready to introduce how we tested these goals.

Chapter 6

Test-driven development

In the implementation chapter, we mentioned that tests were written during implementation and that the TDD technique was used. "Test-driven development is a software development approach in which test cases are developed to specify and validate what the code will do." [21] What this means to us is that before the implementation of any public method, we should write a test for this method. This chapter will focus on implementing the tests and some base principles of test-driven development and writing tests in general.

We need a framework that will help us write and run tests to write and run tests. In .NET platform are number of test frameworks that we can use, here is a list of the most used ones:

- MSTest — Microsoft original test framework
- NUnit — Originaly ported from JUnit [22]
- xUnit — Created by author of the NUnit v2 [23]

Per the author's experience, we will use the xUnit framework, but TDD can be done in any of them.

.NET CLI tool has a template for xUnit project available under command `dotnet new xunit`. This command will prepare a project with dependencies for xUnit and Microsoft.NET.Test.Sdk packages. It will also configure runners and coverlet.collector for collecting code coverage.

6.1 More information about TDD

Test-driven development can be summed up using an activity diagram. At the start of the development, we write down all the tests that we need to cover our API, and then we start with the implementation of the library. We refactor our code; for example, we move code from one method into multiple methods and even classes. After refactoring is done, we check our tests and write new ones.

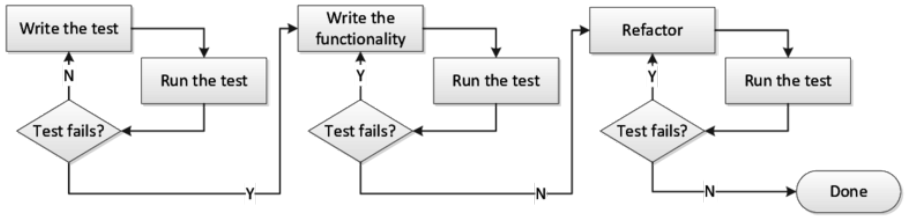


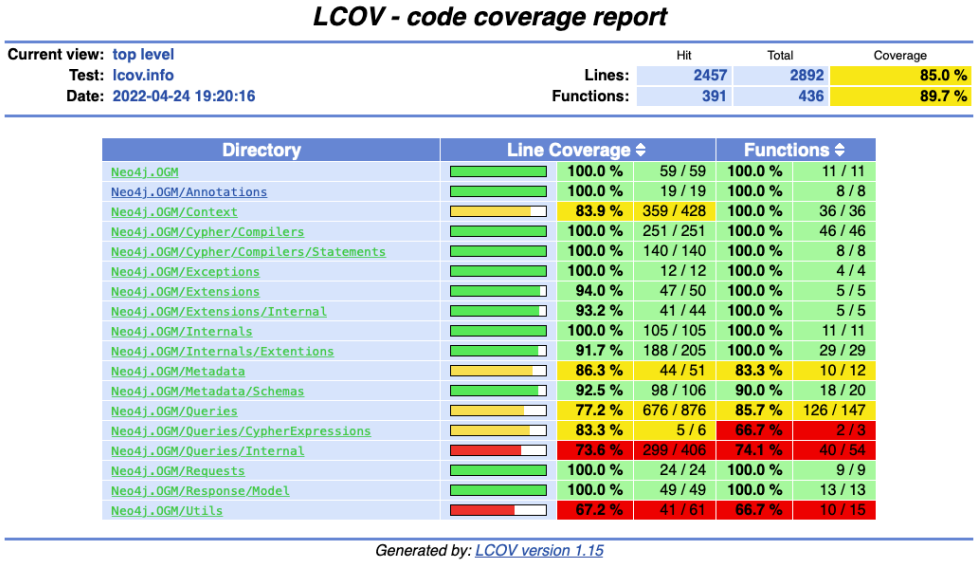
Figure 6.1: Test-driven development diagram [1]

Using this approach has many benefits. One of them is complete code coverage of our code, which means every method should be covered by a test. We will use the coverlet tool to calculate the coverage, a cross-platform coverage framework for .NET. [24] It supports multiple output formats, but for our case, we will use `lcov` format, which can be then loaded by extension in Visual Studio Code and display coverage information directly in files. We can also generate a report of the coverage using these commands.

- ```
  dotnet test \
  /p:CollectCoverage=true \
  /p:CoverletOutput=../../lcov.info \
  /p:CoverletOutputFormat=lcov
  ```
- `genhtml lcov.info -o ./CoverageReport/`

The first command is to execute tests and generate `lcov.info` file, which contains coverage information. This file is also used by extension for Visual Studio Code named Coverage Gutters. The second command is to generate HTML report from `lcov.info` file.
  ```

Figure 6.2 is an example of the report from this project.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	2457	2892	85.0 %
Test:	lcov.info	Functions:	391	436	89.7 %
Date:	2022-04-24 19:20:16				

Directory	Line Coverage		Functions	
Neo4j.OGM	100.0 %	59 / 59	100.0 %	11 / 11
Neo4j.OGM/Annotations	100.0 %	19 / 19	100.0 %	8 / 8
Neo4j.OGM/Context	83.9 %	359 / 428	100.0 %	36 / 36
Neo4j.OGM/Cypher/Compilers	100.0 %	251 / 251	100.0 %	46 / 46
Neo4j.OGM/Cypher/Compilers/Statements	100.0 %	140 / 140	100.0 %	8 / 8
Neo4j.OGM/Exceptions	100.0 %	12 / 12	100.0 %	4 / 4
Neo4j.OGM/Extensions	94.0 %	47 / 50	100.0 %	5 / 5
Neo4j.OGM/Extensions/Internal	93.2 %	41 / 44	100.0 %	5 / 5
Neo4j.OGM/Internals	100.0 %	105 / 105	100.0 %	11 / 11
Neo4j.OGM/Internals/Extentions	91.7 %	188 / 205	100.0 %	29 / 29
Neo4j.OGM/Metadata	86.3 %	44 / 51	83.3 %	10 / 12
Neo4j.OGM/Metadata/Schemas	92.5 %	98 / 106	90.0 %	18 / 20
Neo4j.OGM/Queries	77.2 %	676 / 876	85.7 %	126 / 147
Neo4j.OGM/Queries/CypherExpressions	83.3 %	5 / 6	66.7 %	2 / 3
Neo4j.OGM/Queries/Internal	73.6 %	299 / 406	74.1 %	40 / 54
Neo4j.OGM/Requests	100.0 %	24 / 24	100.0 %	9 / 9
Neo4j.OGM/Response/Model	100.0 %	49 / 49	100.0 %	13 / 13
Neo4j.OGM/Utils	67.2 %	41 / 61	66.7 %	10 / 15

Generated by: LCOV version 1.15

Figure 6.2: Example of the report

We have introduced the concept of TDD and tools for managing tests. Now we will go through the process of their implementation.

6.2 `SessionFactory` tests

We will start with testing `SessionFactory` as it is the entry point to use our library. Tests will be simple, we want to test that `SessionFactory.Create` return an instance of `ISession` and that if the configuration set on creating `SessionFactory` is invalid, that proper exception will be thrown.

6.2.1 Mocking

When writing tests for `SessionFactory`, we encountered a problem with sending assemblies containing our domain model to the `SessionFactory` constructor. We want these assemblies to be mocked.

What is mocking? "Mocking is a process used in unit testing when the unit being tested has external dependencies. The purpose of mocking is to isolate and focus on the code being tested and not on the behaviour or state of external dependencies. In mocking, the dependencies are replaced by closely

controlled replacements objects that simulate the behaviour of the real ones." [25]

For our case, we will use a library for .NET called Moq, which is available as a NuGet package. More information about this package can be found at `https://github.com/moq/moq4`.

The code example 6.1 is of one of the tests for `SessionFactory.Create`, which shows application of mocking. We mock the `Assembly` class, and also set up a mocked resuly of the `Assembly.GetTypes` method. This method is used internally inside the library, and we need to define our result to be able to inject the suitable array of classes against which we want to run the test.

```
1  public void CreateTestResultOk()
2  {
3  var assembly = new Moq.Mock<Assembly>();
4  assembly.Setup(a => a.GetTypes()).Returns(new[] { typeof(Person),
   ↪  typeof(Post) });
5
6  var sessionFactory = new SessionFactory(
7  "connectionString",
8  AuthTokens.Basic("username", "password"),
9  assembly.Object);
10
11 var session = sessionFactory.Create();
12
13 Assert.NotNull(session);
14 Assert.IsAssignableFrom<ISession>(session);
15 }
```

Code 6.1: Example of `SessionFactory` test with mocking of `Assembly` object

6.3 Testing internal classes and methods

In our library, we are using a keyword `internal` which hides classes, methods, or properties from other assemblies, but because we also want to test these classes and their methods, we need to expose them. Exposing `internal` is possible with a slight change in the project file. We need to add the code snippet from the code example 6.2 to the library project file, and our tests will be able to access every internal class and method.

```
<ItemGroup>
 <AssemblyAttribute
 Include="System.Runtime.CompilerServices.InternalsVisibleTo">
 <_Parameter1>Neo4j.OGM.Tests</_Parameter1>
 </AssemblyAttribute>
</ItemGroup>
```

Code 6.2: Snippet of `Neo4j.OGM.csproj` to access internal classes and methods inside test project

We are using this alteration of the `Neo4j.OGM.csproj` to access internal classes and methods inside the test project, we can now prepare a test for internal classes and methods.

6.4 Setup and cleanup

Some of our tests will need to prepare data and mocks. We call this step of tests a setup step. nUnit for example, uses annotations to mark cleanup (`TearDownAttribute`) and setup (`SetUpAttribute`) methods. However, in xUnit, we are using only constructors, and `IDisposable` interface for setup and cleanup, respectively. This approach has a limitation in not being able to perform asynchronous operations safely, but xUnit has a solution for this too. `IAsyncLifetime` is an interface from xUnit that provides the means to perform setup and cleanup methods asynchronously.

Apart from setup and cleanup methods, xUnit also offers an ability to share context between tests, either in one class or across multiple classes using class fixtures or collection fixtures. More on this subject can be found in the official documentation of the xUnit framework. [23]

6.5 Summary

We have now described all the parts of our testing, tests are ready, and we can start with the cycle of implementation and testing as described in TDD diagram in the figure 6.1. This chapter was put after the implementation for clarity reasons.

Chapter 7

Deployment

This chapter will focus on deploying our library to the NuGet repository, GitHub actions, and setting up our repository to attract more contributors.

7.1 NuGet repository

The primary source of packages for .NET applications is `https://www.nuget.org`. Every developer can upload their packages to this repository, and anyone can then search and download packages from this feed.

Before uploading our library to the NuGet repository, we need to set our project properly. We can find out everything we need on the "Package authoring best practices" page in Microsoft's documentation. [26]

From the documentation page [26] we know the properperties we need to set, here is their list:

- `PackageId`: name of the package, which will be used in NuGet repository.
- `Authors`: list of authors of the package.
- `Description`: description of the package.
- `Copyright`: copyright details of the package.

With these properties set, we can now create a registration on NuGet and create an API key for uploading packages using the command line. There is also a possibility to upload a package using the form on the NuGet page itself, but we want to have this process automated using GitHub actions.

7.2 GitHub actions

GitHub actions is a tool for automation of the Continuous Integration/Continuous Delivery (CI/CD) process, which allows us to automate tasks like running tests, checking build status, and publishing packages. It is also possible to run tasks on a schedule, like creating nightly builds of our application or library.

We created three different workflows for our library, each for a different operation in VCS. The operations are:

- push — every push of new commits onto the GitHub server will trigger this workflow.
- pull request — every new pull request to the main branch or update to the pull request will trigger this workflow.
- release — each new tag in the main branch will trigger this workflow.

These workflows will check our library's build status and the execution of the tests we wrote for our library. In the case of release, the workflow will also create a new version of our library and publish it to the NuGet repository.

Workflows also offer us the possibility for static analysis of the codebase. This analysis is helpful as it scans code for common mistakes and potential vulnerabilities. The static analysis is not used in this project, but it is something we can use in the future.

7.3 Setting up repository

One of our main goals with this library is that it will be easy for the community to expand our solution and report any issues with our library. For this reason, we will follow the GitHub community standards. These standards should ensure that anyone can contribute to our project. The list of community standards:

- Description
- README
- Code of conduct
- License

- Issue templates
- Pull request templates

Some of these steps can be generated on GitHub using their templates, like license and code of conduct. Others have to be defined by us. In the repository is a folder called `.github` which contains all the files that we need to set up, except for README and license files which are in the root folder of the repository.

7.4 Summary

With the community standards set up, we have everything we need to release our library. Our repository on the GitHub server can be made public, and we can release our library to the NuGet repository. We can now plan the following steps to finish our library.

Chapter 8

Evaluation of the project

We will now evaluate our project. We can divide this evaluation into three parts.

The first part is the evaluation of our design of the OGM library. The design was inspired by both the Neo4j–OGM for Java implementation and the Entity Framework. The result is a library design, but with some limitations that should be addressed in future development, namely the lack of the change tracker.

The second part of this evaluation is the implementation of a proof-of-concept for the design. We set two goals for the proof-of-concept:

- Create or update nodes in a database
- Map LINQ to Cypher

We achieved these two goals, but we noticed the missing change tracker during the implementation, which severely limits the library's functionality.

The first goal was achieved with the help of an already existing implementation in Java. Parts of the code were translated from Java to C# with some changes to reflect the best practices. This allowed us to quickly develop the save mechanism for our library, although it was not the best solution. The problem with this solution is the different architecture of the library in the Java and .NET. In the .NET, we would greatly benefit from the ability to use dependency injection, which is not used in Java implementation. One thing we could also borrow from the Entity Framework is the change tracker. This

component of the Entity Framework gives us the ability to track changes and save only the changes. It is also implemented using LINQ extensions. The extensions would be easy to implement because we already have some basic functionality in the library.

The second goal was to use the LINQ to create Cypher queries for the Neo4j database. This goal was far more challenging to accomplish than the first one. We had to remap the original expression tree into a new one that could be translated into Cypher query, executed and mapped to the objects using Neo4j driver for .NET. We accomplished this goal with some limitations. These limitations were more about the time and size of the implementation itself than the problem with the design. The only missing part of this goal is to correctly map the relationship between objects and aggregation methods.

The third part of this evaluation is the projects set up for contributors. This part is fully completed. We created a set of GitHub actions for our CI/CD, prepared a contributor's guide and released our library on the NuGet repository feed.

8.1 Next steps

We will propose the next steps for this library and document them in the repository using issues. The next steps should be the following:

- Implement the change tracker.
- Refactor internal classes to use dependency injection.
- Implement the save operation with the use of the change tracker.
- Implement the mapping of the objects using LINQ like it is done in the Entity Framework.

With these steps done, we could release our library as production-ready and could be used in commercial projects.

One other thing missing in this project is proper user-oriented documentation. As this is a proof-of-concept, we did not create any documentation for the library, except for the `README` file. However, there should also be a goal of creating proper documentation for the library for future development.

Conclusion

This thesis aimed to design and create a proof-of-concept for the library for the .NET platform that would be able to map objects into graphs and vice versa with the ability to execute queries on the Neo4j graph database.

At the begging of this thesis, we studied graph databases, their properties and their differences from the relational databases. We also studied the C# language and platform to understand how the LINQ library works.

Understanding both the language and the database system, we looked into similar solutions. The first one was the Entity Framework, which is the ORM library for the .NET platform. The second was the Neo4j–OGM library for the Java.

Having studied both technologies and existing solutions, we defined our goals for the library's design. Based on these goals, we created a design for our library.

Using the design we created, we implemented a proof-of-concept for the library. The proof-of-concept has been tested using TDD principles. This library has limited functionality, but it proves the possibility of the translation between LINQ and Cypher queries. At the end of the implementation, we prepared our project for release by setting up GitHub actions providing us with the CI/CD capabilities.

At last, we evaluated our solution, which met all the proposed goals of implementing the proof-of-concept. However, we pointed out the oversights in our design and proposed the next steps for the library. With these steps implemented, the library should be ready for use in the production environment.

Bibliography

[1] Madeyski, L.; Kawalerowicz, M. Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool. In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, Angers, France: SciTePress - Science and and Technology Publications, 2013, ISBN 978-989-8565-62-4, pp. 260–267, doi:10.5220/0004587202600267. Available from: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004587202600267

[2] Morgante, V. What is a graph database? Apr. 2021, publication Title: Medium. Available from: https://towardsdatascience.com/what-is-a-graph-database-249cd7fdf24d

[3] Cox, G. Introduction to Graph Databases. May 2017, publication Title: Compose Articles. Available from: https://www.compose.com/articles/introduction-to-graph-databases/

[4] Chao, J. Graph Databases for Beginners: Native vs. Non-Native Graph Technology. Dec. 2018, publication Title: Neo4j Graph Data Platform. Available from: https://neo4j.com/blog/native-vs-non-native-graph-technology/

[5] McCreary, D. The Neighborhood Walk Story. Jan. 2021, publication Title: Medium. Available from: https://dmccreary.medium.com/how-to-explain-index-free-adjacency-to-your-manager-1a8e68ec664a

[6] Neo4j. Company. Publication Title: Neo4j Graph Data Platform. Available from: https://neo4j.com/company/

[7] Neo4j. Neo4j Graph Database. Publication Title: Neo4j Graph Data Platform. Available from: https://neo4j.com/product/neo4j-graph-database/

[8] Resources · openCypher. Available from: https://opencypher.org/resources/

[9] Robinson, I.; Webber, J.; et al. *Graph databases*. Sebastopol, CA: O'Reilly, 2015, ISBN 978-1-4919-3200-1. Available from: https://neo4j.com/graph-databases-book/?ref=home

[10] Neo4j. Getting Started with Cypher - Developer Guides. Publication Title: Neo4j Graph Database Platform. Available from: https://neo4j.com/developer/cypher/intro-cypher/

[11] Neo4j. Querying with Cypher - Developer Guides. Available from: https://neo4j.com/developer/cypher/querying/

[12] Neo4j. Updating with Cypher - Developer Guides. Publication Title: Neo4j Graph Database Platform. Available from: https://neo4j.com/developer/cypher/updating/

[13] Hoyos, M. What is an ORM and Why You Should Use it. Dec. 2018. Available from: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a

[14] Albahari, J.; Albahari, B. *C# 8.0 pocket reference: instant help for C# 8.0 programmers*. Sebastapol, CA: O'Reilly Media, Inc, first edition edition, 2019, ISBN 978-1-4920-5121-3.

[15] Rodenburg, J. *Code like a pro in C#*. Shelter Island, New York: Manning Publications, 2021, ISBN 978-1-61729-802-8.

[16] Microsoft. What is .NET? An open-source developer platform. Publication Title: Microsoft. Available from: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

[17] Neo4j. Reference - OGM Library. Publication Title: Neo4j Graph Database Platform. Available from: https://neo4j.com/docs/ogm-manual/3.2/reference/

[18] Neo4j. Client applications - Neo4j .NET Driver Manual. Publication Title: Neo4j Graph Database Platform. Available from: https://neo4j.com/docs/dotnet-manual/4.4/client-applications/

[19] Wagner, B. params keyword for parameter arrays - C# reference. Available from: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params

[20] Wagner, B. internal - C# Reference. Jan. 2022. Available from: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal

[21] Hamilton, T. What is Test Driven Development (TDD)? Tutorial with Example. June 2020. Available from: https://www.guru99.com/test-driven-development.html

[22] Poole, C.; Prouse, R. NUnit.org. Available from: https://nunit.org/

[23] .NET Foundation. Home. Publication Title: xUnit.net. Available from: https://xunit.net/

[24] Coverlet. Apr. 2022. Available from: https://github.com/coverlet-coverage/coverlet

[25] Progress Software Corporation. Mocking Framework for Unit Testing - Telerik JustMock. Available from: https://www.telerik.com/products/mocking/unit-testing.aspx

[26] Gill, C. R. Package authoring best practices. Feb. 2022, publication Title: Package authoring best practices Type: web page. Available from: https://docs.microsoft.com/en-us/nuget/create-packages/package-authoring-best-practices

[27] MongoDB. What Is NoSQL? NoSQL Databases Explained. Publication Title: MongoDB. Available from: https://www.mongodb.com/nosql-explained

Appendix A

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

API Application Programming Interface.

CI/CD Continuous Integration/Continuous Delivery.

CIL Common Intermediate Language.

CLI Command Line Interface.

CLR Common Language Runtime.

DBMS Database Management System.

DIC Dependency Injection Container.

DTO Data Transfer Object.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

JIT Just-In-Time.

NoSQL NoSQL is a term for a database that does not use a relational model to store data. [27].

OGM Object-Graph Mapper.

ORM Object-Relational Mapper.

POJO Plain Old Java Object.

RDBMS Relational Database Management System.

SDK Software Development Kit.

SQL Structured Query Language.

TDD Test-Driven Development.

VCS Version Control System.

Appendix B

Contents of enclosed CD

- `README.txt` ..... the file with CD contents description
- `dotnet-neo4j-ogm` ..... the project's source code directory
  - `src` ..... the directory of source codes
    - `Neo4j.OGM` ..... the project's implementation
  - `tests` ..... the directory of tests
    - `Neo4j.OGM.Tests` ..... the project's tests
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
- `text-src` ..... the thesis text source directory