



Fachhochschule für die Wirtschaft Hannover
- FHDW -
Praxisarbeit

Thema:
Designing a Reflection-Based ORM
Configuration Framework in .NET

Prüfer:
Prof. Dr. Cornelius Köpp

Verfasser:
Frederik Höft (Matr.-Nr. 7015778)

Studiengang HFI420IN

Eingereicht am:
11.01.2023

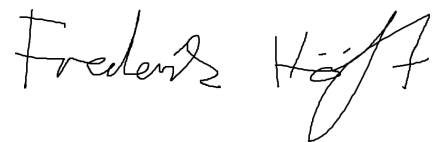
Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe. Alle Textstellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, habe ich mit korrekten Quellenangaben versehen. Über Zitierrichtlinien bin ich schriftlich informiert worden. Diese Arbeit wurde bisher in gleicher oder ähnlicher Form, auch nicht in Teilen, keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Declaration of Authorship

I hereby certify that I have written the present work independently and used no other sources than those indicated. All text passages taken literally or in substance from publications are provided with correct source references. I have been informed in writing about citation guidelines. This work has not been submitted in the same or similar form, not even in part, to any other examination authority and has not been published.

Hannover, 11.01.2023



Ort, Datum

Frederik Höft

Abstract

[Object-relational mapping \(ORM\)](#) is a widely used technique in software development to manage the interaction between relational databases and object-oriented programming languages. It allows developers to work with the database using familiar object-oriented concepts rather than having to write raw SQL queries. However, given the risk of increased maintenance costs and decreased developer productivity posed by inefficient [ORM](#) configurations for large-scale software systems with complex and interconnected business logic, it is vital to ensure that these configurations are as clear and concise as possible.

In the context of this work, the challenges and limitations of existing solutions regarding [ORM](#) in C#/.NET have been analyzed and were translated into requirements. After a conceptual phase, the [Reflective Entity Configuration And Procedure mapping framework \(RECAP\)](#) has been designed and developed as a solution to simplify, standardize, and automate the configuration of relational database access layers in .NET. By building upon the existing functionality of Microsoft's [Entity Framework Core](#), [RECAP](#) streamlines the configuration process for entity mappings and stored database procedures in .NET, resulting in improved maintainability and reduced code redundancy. This is achieved through the use of a uniform configuration syntax and the ability to map procedures to entity-like command objects.

Contents

| | |
|---|-----------|
| Abstract | ii |
| 1 Introduction | 1 |
| 1.1 WKG Software GmbH | 1 |
| 1.2 Goals of this work | 1 |
| 1.3 Structure of this work | 2 |
| 2 Theoretical Foundation | 3 |
| 2.1 The .NET platform | 3 |
| 2.1.1 Common Intermediate Language | 3 |
| 2.1.2 The Reflection API | 3 |
| 2.1.3 Expression Trees | 4 |
| 2.2 Object-relational mapping | 4 |
| 2.3 Entity Framework Core | 5 |
| 2.3.1 Entity mapping | 5 |
| 3 Problem analysis and requirements | 7 |
| 3.1 Data annotations vs. fluent API | 7 |
| 3.1.1 Data annotations | 7 |
| 3.1.2 Fluent API | 8 |
| 3.1.3 Decision for fluent API | 9 |
| 3.2 Missing procedure and function mapping | 9 |
| 3.3 Requirements | 10 |
| 4 Conceptual solution | 12 |
| 4.1 Entity configuration | 12 |
| 4.1.1 The configuration interface | 12 |
| 4.1.2 Physical separation from business logic | 13 |
| 4.1.3 Configuring inheritance | 14 |
| 4.1.4 Entity validation | 16 |
| 4.2 Stored procedure mapping | 17 |
| 4.2.1 Procedure configuration API | 17 |
| 4.2.2 Stored procedure builder API | 18 |
| 4.2.3 Compiler namespace | 19 |
| 4.2.4 Runtime namespace | 21 |
| 5 Implementation and Optimization | 23 |

| | | |
|----------|--|-----------|
| 5.1 | Reflective configuration process | 23 |
| 5.2 | I/O container accessor generation | 23 |
| 5.2.1 | Property accessor optimization process | 24 |
| 5.3 | Requirement validation and testing | 24 |
| 6 | Conclusion | 26 |
| 6.1 | Evaluation | 26 |
| 6.2 | Outlook on further development | 27 |
| 6.2.1 | Handling result collections | 27 |
| 6.2.2 | Backing field resolver | 27 |
| 6.3 | Final thoughts | 27 |
| | List of Figures | 28 |
| | List of Tables | 29 |
| | List of Listings | 31 |
| | Acronyms | 32 |
| | Glossary | 33 |
| | References | 37 |
| | Literature | 37 |
| | Online sources | 38 |
| | Appendix | 42 |

1 Introduction

This applied research project was conducted at FHDW Hannover in collaboration with WKG Software GmbH.

1.1 WKG Software GmbH

WKG Software GmbH is a medium-sized IT service provider and software company based in Hanover, Germany. Established in 2002, the privately held firm specializes in enterprise software solutions, with its flagship products being TelePhone, an enterprise resource planning solution, and SalesFront, an iPad application for media consultants.

[REDACTED], WKG has sought out new customers and developed the web-based SummitCloud solution in 2018. This enterprise platform provides tools for preparing, creating, and generating business reports for listed companies in Germany.

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

To address the increasing size and complexity of their code bases, WKG's development team has commissioned a program to standardize and simplify their Entity Framework Core (EF Core) object-relational mapping (ORM) layer configuration. This will help ensure the reliability and scalability of the company's enterprise software solutions in the years to come.

1.2 Goals of this work

The primary goal of this work is to design and implement a C#/.NET library called RECAP (Reflective Entity Configuration And Procedure mapping framework) that aims to enforce specific conventions and best practices when using EF Core in a project. The purpose of RECAP is to assist developers in writing more organized and maintainable code while also reducing the complexity and redundancy of the configuration process.

Specifically, RECAP aims to reduce the coupling between the EF Core configuration code and the entities themselves, enabling developers to write more flexible and modular code. By using RECAP, developers can more easily change or update the structure of their

entities or the configuration of their database access layer without having to rewrite large portions of their code in multiple locations.

Another objective of [RECAP](#) is to extend the capabilities of [EF Core](#) by providing mechanisms to map stored database procedures to C# types. As a result, the library has been designed to allow developers to utilize familiar object-oriented concepts when working with stored procedures rather than writing raw SQL queries or employing other low-level [application programming interfaces \(APIs\)](#) like [ADO.NET](#) directly.

All of these goals aim to increase productivity and code quality when working with [EF Core](#) by reducing the amount of boilerplate code that developers need to write and maintain, therefore elevating the focus on business logic rather than on the mechanics of interacting with a database.

1.3 Structure of this work

Five additional chapters follow this introduction. Chapter [2](#) establishes a theoretical foundation for the subsequent chapters by introducing the .NET platform, [ORM](#) in general, and [Entity Framework Core](#) specifically. Chapter [3](#) analyzes potential problems of current solutions and derives requirements from them. Chapter [4](#) presents a conceptual solution to address these issues. This solution is then implemented and optimized in chapter [5](#). Finally, chapter [6](#) evaluates and discusses the results of this work.

2 Theoretical Foundation

This chapter provides a theoretical foundation for this work and introduces the reader to the relevant concepts used in the implementation. The first part of this chapter acquaints the reader with the .NET platform, the [Intermediate Language \(IL\)](#), and the concept of reflection. This will serve as a basis for the conceptual design and implementation of the proposed [RECAP](#) framework as well as for the evaluation presented thereafter. The following sections provide a brief introduction to the [ORM](#) paradigm and [EF Core](#), as well as an overview of its current state and limitations. This is necessary since the proposed [RECAP](#) framework will be built upon [EF Core](#).

2.1 The .NET platform

.NET is a platform for building and running applications in a managed virtual machine, making it operating system- and hardware-independent. Additionally, it is designed to be language-independent, allowing a multitude of programming languages to be used (e.g., C#, VB.NET, and F#). In order to allow a single .NET application to run on different platforms, source code is first compiled into byte code, referred to as [Common Intermediate Language \(CIL\)](#), or also just [Intermediate Language \(IL\)](#), which can then be distributed and executed on any platform with a .NET runtime installed [[ECM12](#), pp. 9–12, 290]. A comprehensive introduction to the .NET platform is provided in appendix [A.1](#).

2.1.1 Common Intermediate Language

The [Common Intermediate Language \(CIL\)](#) is the byte code format used by [Common Language Runtime \(CLR\)](#) to execute .NET applications. [IL](#) is a stack-based virtual instruction set that is compiled from the source code of a .NET application. At runtime, the [IL](#) code is then compiled into native machine code by the [just-in-time \(JIT\)](#) compiler [[Kok18](#), p. 238]. In addition to being platform-independent, [IL](#) contains type metadata that is used by the [JIT](#) compiler and the [CLR](#) for optimizations and type checks. This metadata holds information about the types and members of an application, including their names and attributes, and is used to provide the [Reflection API](#) for dynamic inspection of these elements [[Mic21](#)][[ECM12](#), pp. 53, 122–127].

2.1.2 The Reflection API

The [CLR](#) provides the [Reflection API](#) to dynamically inspect the metadata stored in the host process' [IL](#) byte code. This allows the developer to dynamically enumerate and inspect

types, class members, and other elements of the application. Through reflection, developers can dynamically load assemblies, instantiate or inspect types and members, invoke methods, or even emit [IL](#) code for a new method at runtime using the [Reflection.Emit](#) API [ECM12, pp. 53, 440].

2.1.3 Expression Trees

The .NET platform provides a way of representing code as data, referred to as [expression trees](#). This is achieved by representing a code fragment as a tree-like data structure where each node represents an expression such as a method invocation or binary operation. [Expression trees](#) can be dynamically constructed at runtime and subsequently interpreted by the [CLR](#) or compiled into [IL](#) code and then executed as a [delegate](#) [ECM22, pp. 83–84].

The .NET Compiler Platform ([Roslyn](#)) also provides the option to do the opposite, namely to generate [expression trees](#) from a lambda expressions during compilation¹. These compiler-generated [expression trees](#) allow the developer to write an anonymous lambda function and then dynamically inspect the generated [expression tree](#) at runtime. [EF Core](#) performs this dynamic inspection, for example, to generate and execute SQL queries from [LINQ](#) expressions [NET21c], as well as for [fluent API](#) mapping, where [member access lambda expressions](#) are used in client code to specify properties that are reflectively inspected by [EF Core](#) and are then mapped to database columns [NET22c, l. 74][NET22l, ll. 170–171].

2.2 Object-relational mapping

[Object-relational mapping \(ORM\)](#) is a programming technique that enables developers to work with data stored in a relational database using object-oriented programming concepts. [ORM](#) abstracts the database layer of an application, allowing developers to write database engine-agnostic code that interacts with objects rather than handling the complexities of a specific relational database directly. Furthermore, [ORM](#) tools facilitate the mapping between objects in an application and the corresponding data stored in a database. These mappings allow developers to utilize the objects in their code as if they were interacting with physical objects while the [ORM](#) tool translates requests for data modification or retrieval into the appropriate SQL queries and maps returned data back into objects that can be utilized by the application. [ORM](#) can benefit developers by allowing intuitive object-oriented code and the ability to switch between different database technologies effortlessly. However, [ORM](#) may also incur performance penalties compared to raw SQL queries and may result in a discrepancy between the object-oriented

¹ This behavior is demonstrated in appendix [A.1.7](#)

model and the underlying relational database structure [SKS20, pp. 381–382][Che+16, pp. 9–10][NET22p].

2.3 Entity Framework Core

Entity Framework Core (EF Core) is an open-source ORM framework developed by Microsoft that uses reflective programming to implement the Metadata Mapping pattern [NET23][Fow02, pp. 306–309][NET22d]. It is a cross-platform version of its .NET Framework counterpart, Entity Framework, and supports various relational and non-relational databases [NET22e]. EF Core allows developers to either map existing database structures to domain models through the Database-First approach or generate database structures from domain models through the Code-First approach [NET20b].

Entity classes represent database tables or views and contain properties corresponding to columns in the database [NET22j]. While EF Core does allow for the inclusion of business logic in these classes, it does not promote the use of [create, read, update, and delete \(CRUD\)](#) operations within them. Instead, the separate `DbContext` class is provided for interacting with the database and tracking changes to entities (Unit of Work pattern) [NET22f][Fow02, pp. 184–194]. These changes can be saved to the database by calling the `SaveChanges()` method on the corresponding `DbContext` instance, which is specific to the current transactional context and not shared across threads. In addition to providing change tracking, the `DbContext` allows access to `DbSet` instances, which are collections of entities that can be queried and modified (Repository pattern) [NET22f][Fow02, pp. 322–324]. The `DbSet` class implements the `IQueryable<T>` interface and therefore offers a [LINQ-enabled API](#) for querying entities [NET22g, l. 46].

2.3.1 Entity mapping

Prior to querying or modifying entities, the entity classes must be mapped to the corresponding database tables and columns. The following summary² presents the three approaches that can be utilized for entity mapping in EF Core [NET22d]:

Convention-based mapping

[Convention-based](#) mapping is a minimal configuration method for associating entities with database tables. EF Core automatically maps entity classes and properties to database tables and columns based on their names and data types. Therefore, entity classes and properties must adhere to a naming convention that aligns with the database schema.

² A more detailed introduction to the available entity mapping approaches is provided in appendix [A.2](#)

Data annotations

[Data annotation](#) attributes can be applied directly to entity classes and properties for straightforward configuration within the entity classes. These attributes offer more flexibility than [convention-based](#) mapping, such as mapping entity properties to columns with different names, but they are less flexible than [fluent API](#) mapping.

Fluent API

[Fluent API](#) mapping offers the most flexibility among the three mapping approaches. It utilizes a builder pattern, providing a set of methods that can be employed to configure the model. [Fluent API](#) allows for more complex mappings than those achievable with [data annotations](#), as the configuration code is not limited to a single entity or member. This versatility allows for configuring navigation properties, composite keys, and many-to-many relationships.

3 Problem analysis and requirements

This chapter investigates the challenges and issues of using a combination of [fluent API](#) and [data annotations](#) for [ORM](#) configuration that were previously experienced during software development at [WKG](#). Through this problem analysis, requirements for [RECAP](#) are gathered, with the goal of introducing a standardized solution that improves the readability, maintainability, and overall efficiency of [WKG's EF Core](#) entity configuration processes and development workflow.

3.1 Data annotations vs. fluent API

[Fluent API](#) and [data annotations](#) are frequently utilized for model configuration in [EF Core](#) due to their versatility. However, it is essential to consider which approach is most appropriate for standardization within [WKG](#) through the introduction of the [RECAP](#) framework.

3.1.1 Data annotations

[Data annotations](#) provide a simple way to configure entities by allowing configuration attributes to be specified within the class definition. This allows for easy examination of mapped properties and their corresponding [ORM](#) configuration. However, it should be noted that this method may also blend configuration code and business logic, which can decrease readability and result in a cluttered class definition, as illustrated in listing 1.

```
public class UNIT
{
    [Key]
    [StringLength(50)]
    [Unicode(false)]
    public string CODE { get; set; }
    [Required]
    [StringLength(100)]
    [Unicode(false)]
    public string NAME { get; set; }
    [NotMapped]
    public string CodeName => $"{CODE} - {NAME}";
    public virtual ICollection<DATA_ITEM_VALUE> DATA_ITEM_VALUE { get; set; }
}
```

Listing 1: A combination of [data annotations](#) and [convention-based](#) mapping.

Listing 1 illustrates an example of a combined entity configuration using [data annotations](#) and [convention-based](#) naming. In this case, the `UNIT` entity is mapped to a table with the same name in the database. The `CODE` property is designated as the primary key, and the `NAME` property is specified as a required field. [EF Core](#) automatically configures

the navigation property `DATA_ITEM_VALUE` as a collection of `DATA_ITEM_VALUE` objects with a foreign key to `UNIT`. This establishes a relationship between the two entities, with `UNIT` serving as the parent entity and `DATA_ITEM_VALUE` serving as the child entity.

All of the properties in listing 1 are mapped using [convention-based](#) naming, meaning they are mapped to columns with the same name as the property. However, this can pose a problem if a property is renamed, as the corresponding column name may no longer match and break the database mapping. While it is possible to specify the target column name using the [\[Column\]](#) annotation, this is often omitted due to the extra effort required. As a result, enforcement of explicit naming was made a requirement for [RECAP](#).

Examining the `CodeName` property, it can be seen that it is configured as an unmapped property using the [\[NotMapped\]](#) annotation. Since this property is part of the entity's business logic and not present in the database, it must be designated as such. Failing to include the [\[NotMapped\]](#) annotation can result in [EF Core](#) attempting to map the property to a column in the database, leading to a runtime exception when the entity is queried. To prevent this issue, [RECAP](#) must be designed to automatically detect such unmapped properties and configure them accordingly, or to throw an exception during startup if implicit unmapped properties are detected.

3.1.2 Fluent API

[Fluent API](#) allows for creating more complex mappings compared to [data annotations](#). However, this increased flexibility comes at the expense of code readability and maintainability. Because configuration code is not tied to a specific entity or member, it can be challenging to determine which properties are configured in which locations. This indeterminacy can lead to the configuration of entities being dispersed across multiple files, particularly when [fluent API](#) is used in conjunction with [data annotations](#), resulting in a lack of structure and decreased overall maintainability, as demonstrated in listing 2.

```
[Index(nameof(CODE),
    Name = "IDX_BACKGROUND_JOB_TYPE",
    IsUnique = true)]
public class BACKGROUND_JOB_TYPE
{
    [Key]
    public int ID { get; set; }
    [StringLength(50)]
    [Unicode(false)]
    public string DESCRIPTION { get; set; }
    [Required]
    [StringLength(100)]
    [Unicode(false)]
    public string CODE { get; set; }
}

public class BusinessDbContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.Entity<BACKGROUND_JOB_TYPE>(entity => entity
            .Property(e => e.ID)
            .ValueGeneratedOnAdd());
        builder.Entity<BASE64_DATA_PER_LANGUAGE>(entity =>
            entity.HasOne(d => d.LANGUAGE)
                .WithMany(p => p.BASE64_DATA_PER_LANGUAGE)
                .HasForeignKey(d => d.LANGUAGE_ID)
                .OnDelete(DeleteBehavior.ClientSetNull)
                .HasConstraintName("FK_BASE64PL_LANGUAGE"));
        ... // 1000+ lines of code omitted for brevity :)
    }
}
```

Listing 2: Example of an entity (left) configured employing a hybrid of [data annotations](#) in the entity class itself and [fluent API](#) configuration in the `DbContext` class (right).

Listing 2 illustrates the configuration of a model using a combination of [data annotations](#) and [fluent API](#). For example, the `BACKGROUND_JOB_TYPE` entity's primary key is partially configured using the `[Key]` annotation. At the same time, another part of the configuration resides inside the `DbContext` class, as [fluent API](#) is used to specify the value to be auto-incremented by the database engine.

The `OnModelCreating` method is the designated location for applying [fluent API](#) configuration in [EF Core](#), as it is the only point at which a `ModelBuilder` instance is exposed. In the past, this has led to the practice of configuring all entities within this method, resulting in a single, highly complex file with numerous lines of configuration code from various entities, which can be challenging to maintain.

3.1.3 Decision for fluent API

Both approaches to configuring entities have their distinct benefits and drawbacks. [Data annotations](#) are convenient and straightforward to implement, as they allow for the configuration code to be kept within the same file as the entity definition, facilitating easy modification of entity members. However, [data annotations](#) also offer a smaller set of features compared to [fluent API](#) and may lead to a cluttered class definition.

On the other hand, [fluent API](#) provides more flexibility and the ability to handle complex mappings, even though it can be laborious to pinpoint the location in the source where members are configured in its current implementation. Historically, this has led to a lack of structure and readability in the configuration code of several projects.

In order to accommodate more complex mapping situations, it has been determined that all future entity configurations shall utilize the [fluent API](#). So as to maintain consistency, the existing infrastructure will be refactored to align with this decision. [RECAP](#) should introduce the ability to inject the builder instance directly into the entity class, leading to a more structured and organized configuration process. This approach benefits from the flexibility of [fluent API](#) while maintaining proximity to the entity definition and a level of separation between `static` configuration code and instance-specific business logic.

To remedy the need for manual registration of all entities in `DbContext::OnModelCreating()`, it was decided that [RECAP](#) should be able to automatically detect and load all entity classes based on a mutual base class or interface. This would mitigate the necessity to alter multiple files when adding or removing entities from the system.

3.2 Missing procedure and function mapping

[EF Core](#) has presented challenges in the past regarding the configuration and invocation of stored database procedures and functions.

[EF Core](#) does provide some support for mapping stored procedures and functions to entities, but this support is more comprehensive for functions and generally applies only in certain situations. These situations include using procedures for basic [CRUD](#) operations on entities and invoking procedures using raw SQL that return a fully mapped entity. Functions, in contrast, can be mapped to a [CLR](#) method as long as the method does not include [out](#) or [ref](#) parameters. In practice, more complex procedures must be invoked using raw SQL and [ADO.NET](#)-based `DbParameter` objects [[NET22i](#)][[NET22r](#)][[NET22v](#)][[NET20a](#)][[Gor22](#), pp. 324–330].

An in-depth examination of current practices for procedure mapping conducted in appendix [B.1](#) identified the need for a more flexible and efficient method for invoking stored procedures and functions. Consequentially, it was proposed that functions and procedures be mapped to a command pattern interface, enabling the use of more complex parameter types and return values without relying on raw SQL inside of business logic, which is prone to syntax errors and typos and results in strong coupling with a specific database technology. The configuration process for these command entities should be similar, if not identical, to [fluent API](#) entity configuration.

3.3 Requirements

This chapter examined the challenges and limitations of current approaches to [ORM](#) configuration. Based on this investigation, it was concluded that a reflection-based configuration framework ([RECAP](#)) is required to improve the readability, maintainability, and overall effectiveness of entity configuration processes and development workflow. Table [3.1](#) on the following page summarizes the requirements identified in this chapter and includes additional technical aspects necessary for the implementation of [RECAP](#).

Table 3.1: Summary of requirements for [RECAP](#).

| Requirement | Description |
|-------------|---|
| 1 | Enforce explicit naming of mapped tables, views and columns. |
| 2 | Automatically detect unmapped properties and configure them as such or, alternatively, provide a means of throwing an exception during startup if implicitly unmapped properties are detected. |
| 3 | Implement entity configurations using fluent API . |
| 4 | Inject EF Core 's <code>ModelBuilder</code> instance into the entity class itself, allowing for a more structured and organized configuration process. |
| 5 | Separation of entity configuration from the business logic is desired. |
| 6 | Automatically detect and load all entity classes based on a common base class or interface. |
| 7 | Support for table-per-hierarchy (TPH) , table-per-type (TPT) and table-per-concrete-type (TPC) inheritance must be provided. |
| 8 | Stored procedures and functions must be mapped using a similar syntax as offered by fluent API , but the same flexibility as using raw SQL must be provided. |
| 9 | Invocation of stored procedures using RECAP should incur minimal performance overhead compared to direct invocation using raw SQL. |
| 10 | Introduce command pattern-like interfaces for invoking stored procedures and functions using a parameter object to map and pass arguments to RECAP . |
| 11 | The parameter object used for stored procedure fluent API configuration may be a <code>record</code> type using init-only or read-only auto-properties for mapping. RECAP must handle such cases. |
| 12 | RECAP should mostly be database-agnostic, allowing for easy migration to other database engines. |
| 13 | If database-specific features are used, these should be clearly separated into additional libraries to avoid tight coupling of RECAP with a specific database engine. |
| 14 | EF Core must be used as the underlying ORM engine. |
| 15 | RECAP must be operating system independent and as such implemented in C#/.NET 6+. |
| 16 | Compatibility between RECAP and existing project infrastructure and existing EF Core configuration code must be ensured. |

4 Conceptual solution

The previous chapter outlined the challenges with configuring entities and stored procedures identified in [WKG](#)'s existing [EF Core](#) projects and established the requirements for the [RECAP](#) framework. This chapter presents a conceptual solution that utilizes .NET's reflection capabilities to address these requirements. The proposed solution includes an improved process for configuring entities, as well as a proposal for mapping stored procedures and functions to .NET objects.

4.1 Entity configuration

This section proposes a design architecture for configuring entities in [RECAP](#) to meet the requirements identified in [chapter 3](#).

4.1.1 The configuration interface

According to [requirement 4](#), all configurations must be performed within the entity class. Therefore, as a solution, an interface with a **static abstract** configuration method is proposed that must be implemented by each entity class, as illustrated in [figure 4.1](#).

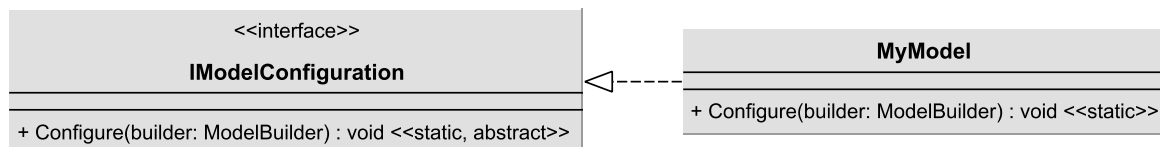


Figure 4.1: Enforcing type-bound configuration methods in [EF Core](#) models.

With the release of .NET 7 in November 2022, the [ECMA-335 CLI Specification Addendum](#) permits the use of previously illegal **IL** metadata patterns, including the ability to define **static abstract** members in interfaces. This allows for the creation of a standardized [API](#) for entity configuration by defining a **static abstract** `Configure()` method in the interface that each entity class must implement. Since entity configuration is specific to a type rather than an instance, it is appropriate for the configuration method to be static. Additionally, since .NET does not permit static methods to be invoked on instances, this ensures logical separation from instance-specific business logic ([requirement 5](#)).

[Figure 4.1](#) illustrates how entities can implement an [API](#) that accommodates the injection of a `ModelBuilder` instance. The implementing entity `MyModel` must then instantiate the corresponding `EntityTypeBuilder<MyModel>` using the generic `ModelBuilder::Entity<T>()` factory

method and configure itself. As the `static abstract` modifier combination is only valid on interfaces, implementations of `Configure()` must be concrete and, therefore, cannot be overridden by child classes [NET21b]. This makes the `IModelConfiguration` interface a suitable entry point for reflective type enumeration and subsequent automatic configuration, as it only requires direct implementations of the interface to be considered. Renaming `IModelConfiguration` to `IReflectiveModelConfiguration` emphasizes the intended use of the interface.

However, further encapsulation is desired to prevent potential unexpected behavior caused by the ability to access and modify the configuration of other entities using the `ModelBuilder`. To ensure that the correct `EntityTypeBuilder<T>` is injected directly into an entity of type `T` and that this entity implements a `Configure()` method with a matching signature, type information about the entity must be provided to both the `IReflectiveModelConfiguration` interface and the [RECAP](#) framework.

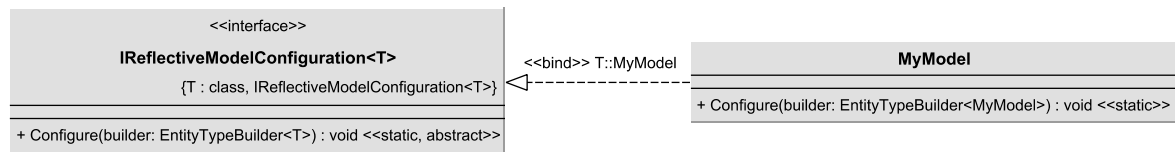


Figure 4.2: Using generics to enforce type-safety and encapsulation.

Figure 4.2 illustrates how generics in the `IReflectiveModelConfiguration` interface can provide type safety and encapsulation. The generic parameter `T` exposes type information to upstream components, enabling the [RECAP](#) framework to construct the `EntityTypeBuilder<T>`. In addition, generic type constraints are utilized to ensure that implementations of `IReflectiveModelConfiguration<T>` are of type `T` themselves, thereby guaranteeing that the `EntityTypeBuilder<T>` passed to the `Configure()` method will match the type of the implementing entity.

4.1.2 Physical separation from business logic

The `Configure()` method is static, which ensures logical separation of the entity configuration from business logic. Although requirement 5 does not specify the desired degree of separation, `partial` classes can be introduced to further achieve physical isolation of concerns. Separating the business logic and configuration code physically facilitates the inspection and maintenance of the entity configuration and business logic, respectively. Furthermore, both files can be placed in the same directory for easy navigation and can be organized using Visual Studio's file nesting feature for convenience. This approach is illustrated in figure 4.3 on the next page, with all business logic placed in `MyModel.cs` and configuration code placed in `MyModel.config.cs`.

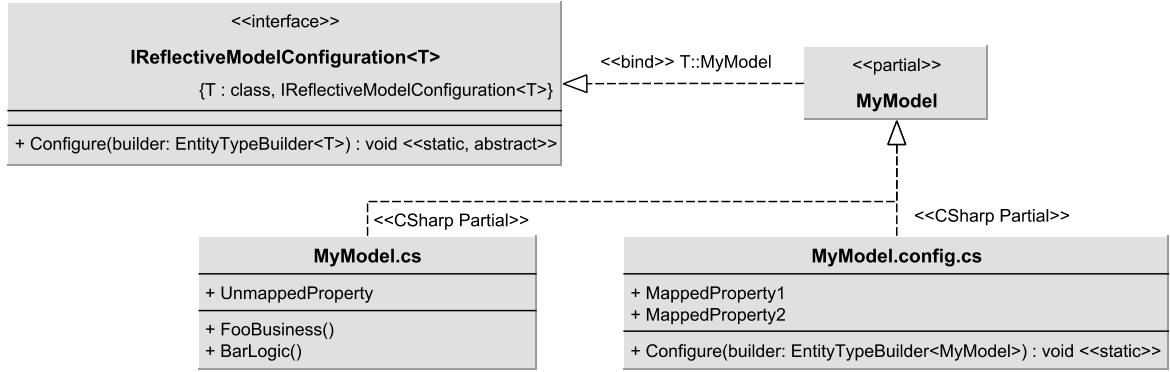


Figure 4.3: Physical separation of configuration from business logic using **partial** classes.

4.1.3 Configuring inheritance

Requirement 7 specifies that the introduction of RECAP must not limit the support for inheritance offered by EF Core, namely [table-per-hierarchy \(TPH\)](#), [table-per-type \(TPT\)](#), and [table-per-concrete-type \(TPC\)](#).

Table-per-hierarchy (TPH)

[TPH](#) is the default inheritance mapping strategy offered by EF Core. It requires the union of all entity properties in the inheritance hierarchy to be mapped to a single table, resulting in a polymorphic table structure. Moreover, an additional discriminator column is introduced to store the concrete entity type [\[NET22m\]](#).

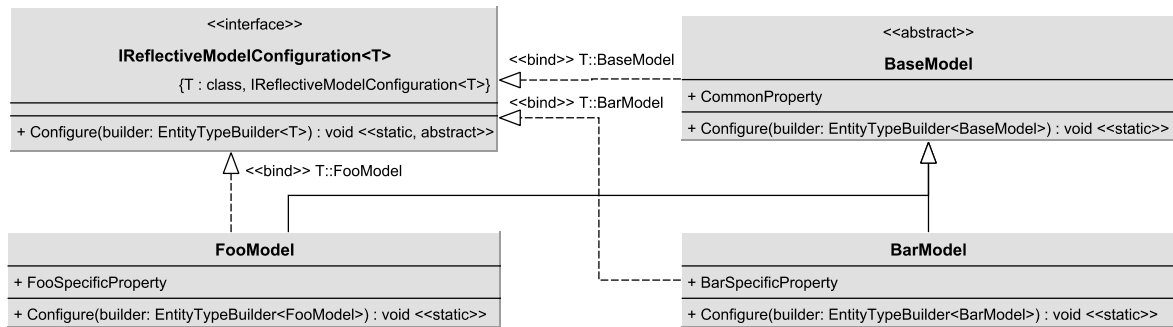


Figure 4.4: The [TPH](#) mapping strategy in RECAP.

Figure 4.4 demonstrates the configuration of [TPH](#) mapping in RECAP. The `Configure()` method in the `BaseModel` class is used to map common properties of the inheritance hierarchy and configure the table's discriminator column. The `FooModel` and `BarModel` classes may also implement their own `Configure()` method to map any additional, type-specific properties. EF Core automatically handles shared properties based on the `EntityTypeBuilder<BaseModel>` configuration.

Table-per-type (TPT)

TPT is a design pattern that simulates inheritance through the use of delegation. Each entity is mapped to its own table, which includes a foreign key referencing its parent entity. This results in a relational table structure that allows shared properties to be defined in the base class while each child class maps its own specific properties [NET22m].

In **RECAP**, **TPT** configuration follows the same class structure as shown in figure 4.4 on the preceding page, but the `Configure()` methods reference a set of tables rather than a single table. The base class `Configure()` method maps the base table, while the `Configure()` methods for child classes specify their respective tables and any additional properties to be mapped. This causes **EF Core** to automatically generate **INNER JOIN** clauses for **SELECT** operations on the base type and to delegate modifying operations to the appropriate table(s) based on the entity's type.

Table-per-concrete-type (TPC)

TPC is a design pattern that maps all declared and inherited properties of each concrete class to its own table, leading to duplicate columns for shared properties. Unlike **TPH**, **TPC** does not map the base class, so shared properties must be configured separately in each subclass [NET22m].

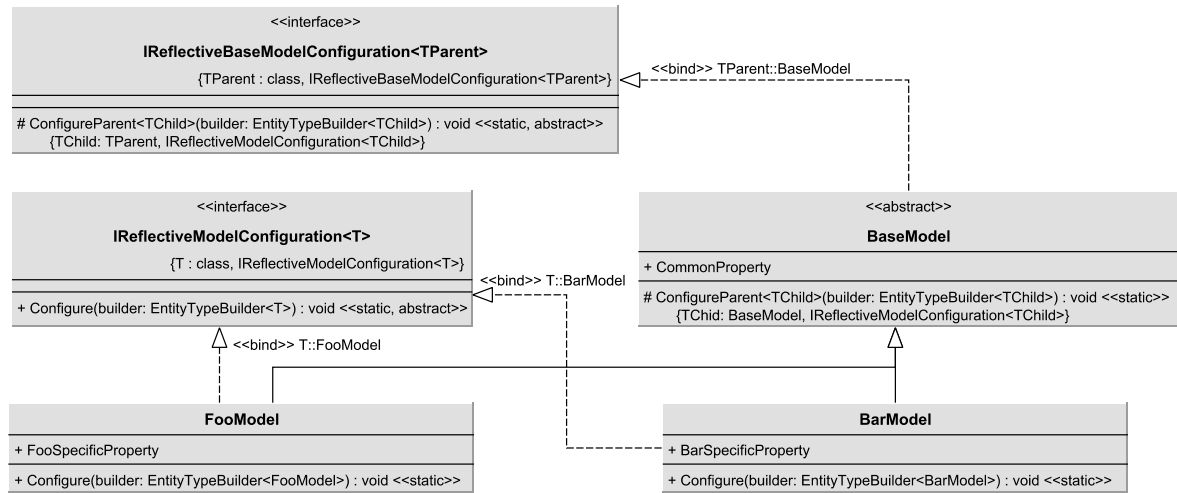


Figure 4.5: The **TPC** mapping strategy in **RECAP**.

Figure 4.5 illustrates how **RECAP** introduces the `IReflectiveBaseModelConfiguration<TParent>` interface to configure shared properties in the base class. The `ConfigureParent<TChild>()` method in this interface must be generic, with the constraint that `TChild` extends `TParent`, because `EntityTypeBuilder<TChild>` is **invariant** on `TChild`. Using a generic method alleviates this limitation, and the configuration of shared properties in the base class can be reused in each subclass.

4.1.4 Entity validation

[RECAP](#) allows the configuration of validation policies for entities. These policies are enforced once when models are being loaded and ensure compliance with requirements [1](#) and [2](#). Additional custom policies may also be specified and enforced using [RECAP](#).

Naming policies

Naming policies can be implemented through the `INamingPolicy` interface, which provides the `Audit()` method for inspecting and validating configured entities and their mapped properties. To fulfill requirement [1](#), the following default implementations are provided:

- `EntityNamingPolicy.AllowImplicit` does not validate whether column and table names were explicitly configured and allows [EF Core](#) to fall back on [convention-based](#) mapping.
- `EntityNamingPolicy.PreferExplicit` is the default strategy. It checks column and table names for explicit configuration and logs warning messages for any that have not been explicitly configured.
- `EntityNamingPolicy.RequireExplicit` validates column and table names and throws an exception if any implicitly-named column or table is encountered.

Mapping policies

`IMappingPolicy` implementations also require the implementation of an `Audit()` method. Mapping policies define the action to be taken when properties are discovered that are neither explicitly mapped nor explicitly ignored:

- `MappingPolicy.AllowImplicit` does not validate properties.
- `MappingPolicy.PreferExplicit` is the default strategy. Properties are validated, and warning messages are logged for each property that has not been explicitly mapped or ignored. No further actions are taken.
- `MappingPolicy.RequireExplicit` validates properties and throws an exception during startup if any property is encountered that has not been explicitly mapped or ignored.
- `MappingPolicy.IgnoreImplicit` validates properties and explicitly ignores any that have not been explicitly mapped or ignored previously.

`MappingPolicy.RequireExplicit` and `MappingPolicy.IgnoreImplicit` satisfy requirement [2](#) by ensuring that no runtime exceptions occur due to unmapped properties.

4.2 Stored procedure mapping

This section presents an architectural solution to address the absence of support for mapping stored procedures and functions in [EF Core](#), which was highlighted in section 3.2. As illustrated in figure 4.6, the `ProcedureMapping` implementation in [RECAP](#) is divided into four primary components: `Infrastructure`, `Builder`, `Compiler`, and `Runtime`.

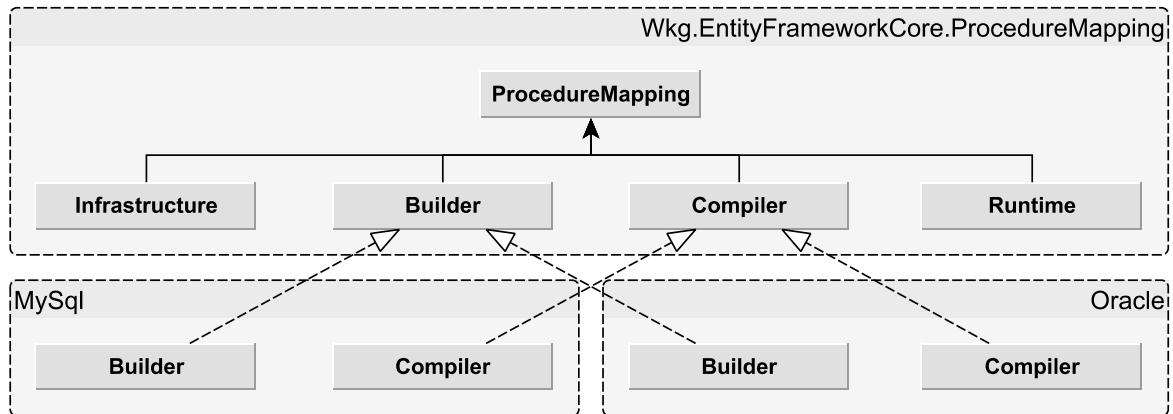


Figure 4.6: Proposed namespace and sub-library structure for stored procedure mapping.

The `Infrastructure` namespace defines the core interfaces used by [RECAP](#)'s [procedure configuration API](#), including the classes that initiate the procedure loading process. The `Builder` namespace offers a fluent builder pattern for configuring and mapping stored database procedures and functions to [procedure command object \(PCO\)](#) classes which serve as [RECAP](#)'s counterpart to [EF Core](#)'s entity classes. The `Compiler` namespace employs reflection and code generation to compile these mapped bindings into executable code, while the `Runtime` namespace manages the [PCO](#) instances and provides access to the dynamic components generated by the `Compiler` namespace. Client code can subsequently use [PCOs](#) for procedure invocation.

Since database engines offer different data types and syntax for stored procedures and functions, concrete implementations of the `Builder` and `Compiler` namespaces are provided in separate assemblies for each supported database engine.

4.2.1 Procedure configuration API

The [procedure configuration API](#) defines the class structure that must be implemented by client code when using [RECAP](#). [RECAP](#) aggregates the parameters of a stored procedure into a [PCO](#)-specific parameter object which can be configured in a similar manner to entities in [EF Core](#). This parameter object, referred to as the [I/O container](#), is used to pass arguments from client code to [RECAP](#) and retrieve any results of the

stored procedure upon invocation. To facilitate this tight coupling, an interface called `IReflectiveProcedureConfiguration<TProcedure, TIOContainer>` has been defined for the configuration of stored procedures, similar to the `IReflectiveModelConfiguration<T>` defined in subsection 4.1.1.

Stored procedures and functions are configured using a builder object, similar to the `EntityTypeBuilder<T>` in [EF Core](#). This `ProcedureBuilder<TProcedure, TIOContainer>` is passed to the `Configure()` method of the implementing [PCO](#), allowing for the configuration of the procedure and its distinct [I/O container](#) using syntax similar to [fluent API](#), as specified in requirement 8.

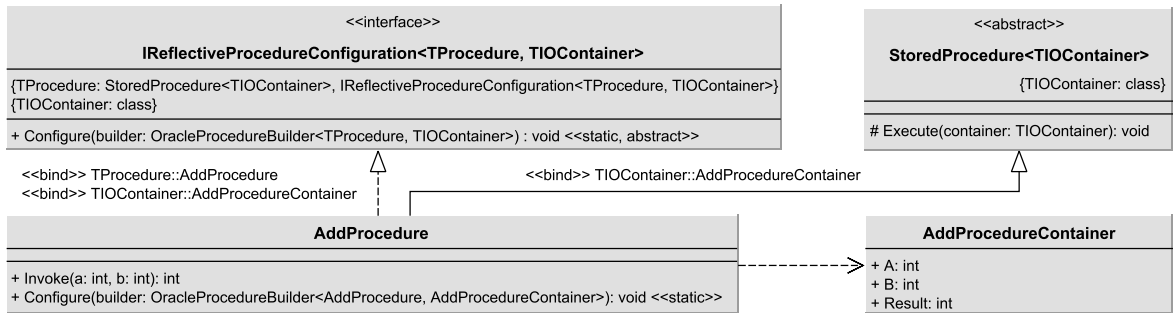


Figure 4.7: Procedure configuration API implementation of a stored procedure targeting PL/SQL.

Figure 4.7 illustrates the configuration of a prototypical [PCO](#) (`AddProcedure`) and the corresponding [I/O container](#) (`AddProcedureContainer`). The `IReflectiveProcedureConfiguration<,>` interface includes generic constraints mandating that the implementing [PCO](#) is of type `TProcedure` and that the [I/O container](#) object, `TIOContainer`, is a reference type³. Additionally, `TProcedure` must derive from `StoredProcedure<TIOContainer>`, which is the [PCO](#) base class and exposes the protected `Execute(TIOContainer)` method for accessing [RECAP](#)'s internal runtime [API](#). By disconnecting the mapped [I/O container](#) from the [PCO](#) itself, the [PCO](#) may include any number of `Invoke()` methods with various signatures to interface with client code. This facilitates a more intuitive and flexible [API](#) for clients while the actual [I/O container](#) remains within the scope of the [PCO](#) and [RECAP](#).

4.2.2 Stored procedure builder API

To facilitate the mapping of stored procedures across various database engines, [RECAP](#) utilizes a combination of universal builder classes that provide shared functionality, and database-specific builders that provide options tailored to a particular engine. Common base classes and interfaces allow for the definition of shared characteristics that apply

³ The `class` constraint in figure 4.7 ensures that [I/O container](#) objects can only be passed by reference, allowing results to be returned via the same instance.

to every [PCO](#), such as the name and type (i.e., function or procedure). At the same time, concrete implementations provide additional settings unique to each database (e.g., the `InPackage(string)` method for specifying the PL/SQL package to be used). Concrete procedure builder classes also include factory methods for creating concrete parameter builders, which utilize [member access lambda expressions](#) and reflection to gather metadata about a specific property of the [I/O container](#) being mapped⁴ and subsequently provide a fluent syntax to bind this property to a matching parameter of the database procedure. An example of the proposed configuration syntax is provided in listing 3.

```

1 public class AddProcedure : OracleStoredProcedure<AddProcedureContainer>,
2     IReflectiveProcedureConfiguration<AddProcedure, AddProcedureContainer>
3 {
4     // ...
5     public static void Configure(OracleProcedureBuilder<AddProcedure, AddProcedureContainer> self)
6     {
7         // PCO mapping options
8         _ = self.ToDatabaseProcedure("proc_add") // specify type (procedure) and name ("proc_add")
9             .InPackage("test"); // specify PL/SQL package ("test")
10        // map properties of the I/O Container object to parameters of the procedure
11        _ = self.Parameter(param => param.A) // use member access expression to specify property
12            .HasName("a") // ... A to be mapped to parameter "a" with ...
13            .HasDbType(OracleDbType.Int32) // ... PL/SQL type INTEGER ...
14            .HasDirection(ParameterDirection.Input); // ... which is also an IN parameter.
15        // ...
16    }
17 }
18 // use records to reduce boilerplate code
19 public record class AddProcedureContainer(int A, int B, int Result);

```

Listing 3: Excerpt of the proposed configuration syntax for the example shown in figure 4.7.

Listing 3 illustrates how the PL/SQL `AddProcedure` [PCO](#) from the example introduced in figure 4.7 on the previous page can be configured using the `OracleProcedureBuilder<,>` class. [RECAP](#) advocates for the use of `record` classes for [I/O containers](#) due to their concise declaration syntax for [init-only](#) properties, even though this requires [RECAP](#) to bypass [init-only](#) restrictions at runtime when updating return value properties in the [I/O container](#). In line 11, a [member access lambda expression](#) is used to create an `OracleParameterBuilder<TIOContainer, TProperty>` instance for the "A" property of the `AddProcedureContainer` [I/O container](#). This property is then configured to be mapped to an [IN](#) parameter named "a" of PL/SQL type `INTEGER`.

4.2.3 Compiler namespace

The `Compiler` namespace is responsible for transforming the mappings defined using builder objects into a type structure that can be used to invoke the mapped procedures and functions at runtime. The namespace can be divided into two parts: classes directly contributing to the build process and those representing the output of a successful build.

⁴ This process is explained in more detail in appendix [A.1.7](#).

Build process

Figure 4.8 visualizes the two-phase build process used by [RECAP](#).

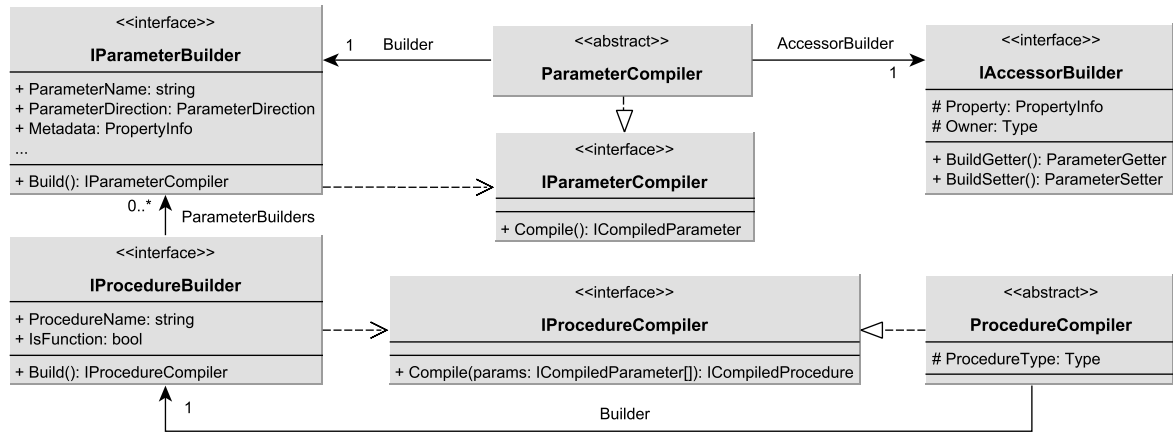


Figure 4.8: Proposed type structure for the `Compiler` namespace using a split build process.

In the first phase of the build process, the method `IProcedureBuilder::Build()` is called, which validates the configured mapping and recursively builds associated `IParameterBuilder` instances into `IParameterCompiler` objects. Finally, the `IProcedureCompiler` object is created and returned.

In the second phase, all `IParameterCompiler` objects are used to create corresponding, stateless `ICompiledParameter` objects. During this process, the property metadata collected from the [member access lambda expression](#) during parameter builder instantiation is used in conjunction with an `IAccessorBuilder` to generate getters and, depending on the direction of the parameter, setters for the mapped [I/O container](#) property. These generated getters and setters will be used at runtime to read and update the [I/O container](#) object's state. Once all parameters have been compiled, they are used to create a concrete `ICompiledProcedure` object, which is then added to the runtime `ProcedureCache`.

Compiler output

As shown in figure 4.9, the compiler output consists of three different categories: compiled procedures, compiled parameters, and parameter accessor [delegates](#).

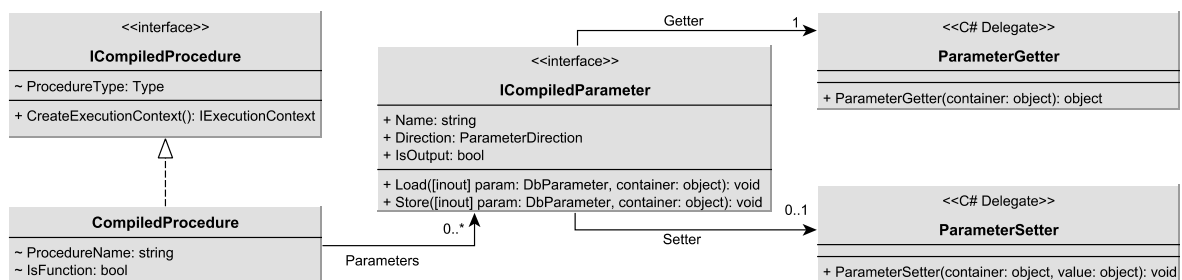


Figure 4.9: Output types of the build process

The `ICompiledProcedure` interface represents a stored procedure or function as an abstract, stateless concept. It provides a factory method for creating the corresponding [execution context](#), which serves as a facade for interacting with the database and maintains the state of the [PCO](#) during procedure invocation. The `ICompiledProcedure` may reference zero or more, also stateless, `ICompiledParameter` objects, which act as a bridge between database engine-specific [ADO.NET](#) `DbParameter` objects and the mapped properties of the [I/O container](#). The `ICompiledParameter` holds generated `ParameterGetter` and `ParameterSetter` delegates that can read and write to specific properties of the [I/O container](#). Because these accessor [delegate](#) functions are generated at runtime for a specific type and property, they exhibit late and early binding characteristics: Their implementation is unknown at compile time, but they do not rely on reflection, minimize the use of boxing, and require the provided parameters to be of a specific type.

4.2.4 Runtime namespace

The `Runtime` namespace allows client code to invoke stored procedures and functions using the runtime [API](#). This includes creating [PCOs](#) from cached `ICompiledProcedure` instances, as well as using the underlying [ADO.NET](#) providers to create and execute database commands.

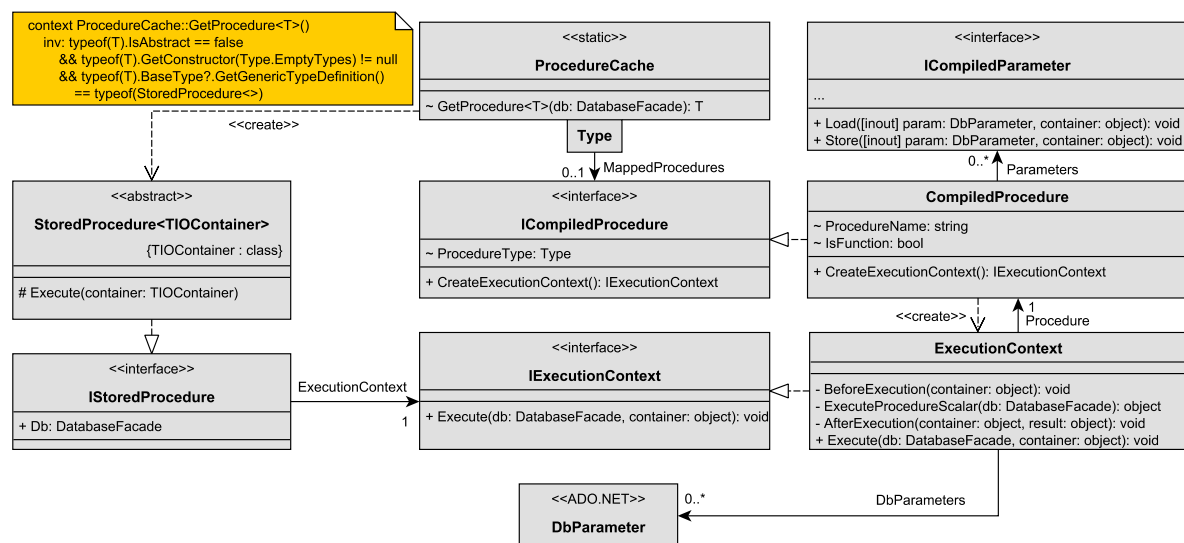


Figure 4.10: The `ProcedureCache` is used to create concrete [PCO](#) instances of generic type `T`.

Figure 4.10 illustrates the structure of the `Runtime` namespace. The `ProcedureCache` class is a singleton that stores `ICompiledProcedure` instances using a dictionary, with the type of the corresponding [PCO](#) serving as the key. The procedure cache can be accessed through a static factory method that constructs the matching [PCO](#) of generic type `T` from the corresponding compiled procedure and the [execution context](#). Generic constraints ensure

that only concrete [PCO](#) types with a public parameterless constructor are provided to the `GetProcedure<T>()` factory method. A concrete [PCO](#) may call the protected `Execute()` method inherited from the [PCO](#) base class, `StoredProcedure<>`, to interact with the runtime [API](#). This call is then forwarded to the underlying [execution context](#) by [RECAP](#). The [execution context](#) populates an [ADO.NET](#) `DbParameter` array with values extracted from the provided [I/O container](#) object using the accessor [delegates](#) in the `BeforeExecution()` method. The use of dynamically generated accessor [delegates](#) is necessary to perform late-bound property access on [I/O container](#) objects, as the concrete type of these objects is only determined at runtime. The `ExecuteProcedureScalar()` method then creates and executes the actual `DbCommand` on the database using the provided database connection. Finally, the `AfterExecution()` method writes any [INOUT](#), [OUT](#), or return values of the procedure back into the [I/O container](#), again using the accessor [delegates](#).

5 Implementation and Optimization

All proposed concepts have been implemented in a [RECAP](#) prototype. This chapter presents key implementation details, including optimizations to enhance performance, and verifies that all requirements have been met.

5.1 Reflective configuration process

As discussed in subsection [4.1.1](#), the `IReflectiveModelConfiguration<T>` interface is used to isolate the configuration of different entities. To initiate the configuration process, an instance of the `EntityTypeBuilder<T>` class that corresponds to the desired entity must be injected into the `Configure()` method in the entity class. [RECAP](#) offers the `LoadReflectiveModels()` extension method on the `ModelBuilder` class as the entry point for the configuration process. This method has optional `INamingPolicy` and `IMappingPolicy` parameters that can enforce naming and mapping conventions for configured entities as mandated by requirements [1](#) and [2](#).

To initiate the configuration process, `LoadReflectiveModels()` must be manually invoked in `DbContext::OnModelCreating()` at startup. The method begins by reflectively enumerating all direct implementations of the `IReflectiveModelConfiguration<T>` interface that are loaded by the executing assembly and then constructs and injects the corresponding `EntityTypeBuilder<T>` instances into the `T::Configure()` methods. Subsequently, [RECAP](#) iterates up the inheritance hierarchy, invoking `ConfigureParent<TChild>()` methods on base classes implementing `IReflectiveBaseModelConfiguration<TParent>` using the same `EntityTypeBuilder<T>` instance to support [TPC](#) hierarchies. The configuration process concludes with the validation of the configured models, during which the provided policies are enforced once the root type (`System.Object`) has been reached.

A more detailed description of this injection process is provided in appendix [C.1](#).

5.2 I/O container accessor generation

Unlike the configuration process, property access of the procedure's [I/O container](#) is a runtime operation that requires optimization to minimize performance overhead. All [I/O containers](#) must be treated as simple [object](#) instances at runtime, as their type signatures may not overlap. As a result, late binding is used to access the mapped properties. Several iterations were required to find an optimal approach that balances performance and flexibility.

The following summary⁵ discusses the iterative development and optimization process of the [I/O container](#) accessor generation.

5.2.1 Property accessor optimization process

This optimization process aimed to provide a lightweight property accessor implementation that can be built from `PropertyInfo` metadata. The first approach utilized the built-in `PropertyInfo::GetValue()` and `PropertyInfo::SetValue()` methods, which rely on reflection to retrieve compiler-generated getter and setter methods. However, this approach was found to be inefficient as it resulted in a significant performance overhead. In addition, the [Reflection API](#) only supports writable and [init-only](#) properties, which did not meet requirement 11. To overcome this limitation, the second approach involved setting the compiler-generated backing field of the [auto-property](#) rather than the property itself through reflection. While this approach successfully supported read-only properties, it still suffered from performance issues.

The third approach utilized [expression trees](#) to emit type-safe, early-bound property accessors by compiling the [expression trees](#) into [IL](#) instructions at runtime. While this method achieved near-perfect performance for get-accessors, set-accessors still required late binding using reflection due to read-only access restrictions.

The final approach involved using the [Reflection.Emit API](#) to generate [IL](#) byte code that directly reads and writes to the property, bypassing access validation and eliminating the dependency on reflection. As demonstrated in appendix D.2, this solution was able to match the performance of direct property access with perfect type knowledge while still providing the flexibility of dynamically determining the concrete implementation at runtime. The implementation generated through this approach is nearly indistinguishable from one emitted by [Roslyn](#) during the compilation of normal [auto-properties](#) in terms of performance and runtime behavior. In order to set the value of a property using this method, a `DynamicMethod` instance is created with two object parameters, with the first representing the target instance and the second representing the value to be assigned. The `ILGenerator` is then used to emit [IL](#) instructions that load the arguments onto the stack, cast and unbox them, and assign the unboxed value to the backing field in the target [I/O container](#) instance.

5.3 Requirement validation and testing

The proposed conceptual solution was implemented in a pre-production prototype of [RECAP](#) and tested against the requirements identified in section 3.3. The results of the requirement validation are presented in table 5.1 on the following page.

⁵ Further implementation details are provided in appendix C.2

Table 5.1: Validation of requirements.

| Requirement | Validation |
|-------------|---|
| 1 | Multiple naming policies are provided by RECAP to enforce explicit naming (see subsection 4.1.4). |
| 2 | Multiple mapping policies are provided by RECAP to prevent implicit convention-based property mapping (see subsection 4.1.4). |
| 3 | Usage of fluent API is encouraged by automatically provided the correct builder instance in the entity class itself (see subsection 4.1.1). |
| 4 | The ModelBuilder instance is injected into the entity class itself allowing in-model fluent API configuration (see subsection 4.1.1). |
| 5 | Logical separation of the configuration from business logic is achieved using static configuration method. Physical separation can be accomplished using partial classes (see subsection 4.1.1 and 4.1.2). |
| 6 | Reflectively loaded interfaces are provided for entities and stored procedures, as well as corresponding extension methods to bootstrap the RECAP framework (see subsection 4.1.1 and section 5.1). |
| 7 | Support for TPH , TPT and TPC inheritance is provided (see subsection 4.1.3). |
| 8 | Stored procedures and functions are configured using fluent syntax (see subsection 4.2.2). |
| 9 | Invocation of stored procedures using RECAP incurs minimal performance overhead compared to direct invocation using raw SQL. This was confirmed by the benchmarks in appendix D.3. |
| 10 | Command objects (PCOs) can be configured, mapped and invoked, representing stored procedures and functions (see subsection 4.2.1 and 4.2.4). |
| 11 | RECAP is able to write to read-only properties of a mapped I/O container . This is achieved using raw IL code, bypassing access validation checks (see subsection 5.2.1 and appendix C.2.3). |
| 12 | RECAP is mostly database-agnostic, allowing for easy migration to other database engines (see section 4.2). |
| 13 | Database-specific features are clearly separated into additional libraries to avoid tight coupling of RECAP with a specific database engine (see section 4.2). |
| 14 | EF Core is used as the underlying ORM provider. |
| 15 | RECAP is operating system independent and implemented in .NET 7 (\implies .NET 6+). |
| 16 | Compatibility with existing project infrastructure and existing EF Core configuration code is ensured as only classes implementing the corresponding reflective interface are loaded. Other entities must be configured manually (see subsection 4.1.1). |

6 Conclusion

This study examined the issues and limitations of current approaches to [ORM](#) configuration. The investigation identified a set of requirements for a reflection-based configuration framework to enhance the readability, maintainability, and overall effectiveness of entity configuration processes and development workflow. A conceptual solution leveraging .NET's reflection capabilities to implement these requirements was proposed. The proposed concepts were subsequently implemented in the [RECAP](#) pre-production prototype, and the framework was analyzed and optimized for improved runtime performance.

6.1 Evaluation

The [RECAP](#) prototype was developed using the latest .NET 7 SDK and C#11.0 as a company-internal set of libraries. To evaluate the performance and functionality of [RECAP](#), a series of benchmarks was conducted while functionality was verified using a variety of test datasets and database configurations.

Benchmark results indicate that the overhead of using [RECAP](#) is minimal compared to manually invoking stored database procedures (see appendix [D.3](#)). In particular, the mean execution time for stored procedure calls using [RECAP](#) was only 3.7% (6.1µs) longer than the average execution time for manual [ADO.NET](#)-based calls. These results suggest that the performance impact of using [RECAP](#) is negligible compared to network latency when interacting with remote database engines, which is often an order of magnitudes larger. Furthermore, as the overhead of [RECAP](#) is constant for a given number of procedure parameters, its significance decreases with increasing complexity and execution time of stored procedures.

[RECAP](#) has already been demonstrated to be viable in a small test application where it successfully interacted with multiple database engines. The test scenario included mapping [TPT](#) and [TPC](#) inheritance hierarchies in SQLite and invoking stored procedures and functions in MySQL and PL/SQL. In addition, an earlier PL/SQL-only development version of the, at the time, still non-reflective framework was successfully deployed in a production environment and is currently being used to map and invoke stored procedures in the ██ project. While at present only used at a small scale, the framework has proven reliable in a production environment and has received positive feedback from the [WKG](#) development team regarding improved code quality and maintainability.

It was concluded that large-scale integration of [RECAP](#) into existing projects has the potential to significantly reduce development time for new database-related features and

maintenance efforts of the existing code base. However, in these cases, substantial structural changes to the code base may be required for a complete migration to [RECAP](#). Overall, [RECAP](#) requires an upfront investment of time and effort for integration into existing projects, but it can greatly benefit the development workflow in the long run, necessitating evaluation on a per-project basis. Nonetheless, as no adverse effects could be determined, it is recommended that consideration be given to incorporating [RECAP](#) into the design and development of any new project initiatives in order to fully capitalize on its capabilities.

6.2 Outlook on further development

The [RECAP](#) prototype is currently undergoing extensive testing by [WKG](#)'s development team. Based on these findings, further improvements to the framework may be made.

6.2.1 Handling result collections

[RECAP](#)'s stored procedure implementation currently only returns the first primitive result row if a return value is configured. This functionality could be expanded by creating a data reader to read multiple rows, allowing collections to be mapped as return values.

6.2.2 Backing field resolver

As discussed in appendix [C.2.1](#), the current setter generation implementation relies on locating a compiler-generated backing field for [auto-properties](#) by name to set the value of read-only properties. However, the naming of these fields is not standardized in the [ECMA-334](#) specification and may vary between different compilers or even change in future releases of the [Roslyn](#) compiler. The community has made attempts to address this issue, such as using the [IL](#) pattern of the compiler-generated backing field getter method to extract field metadata [[EZ20](#)].

Further investigation is required to determine the viability and necessity of such a resolver.

6.3 Final thoughts

The [RECAP](#) prototype has been made available for preview on [WKG](#)'s internal Nuget package server. All requirements have been met, and the prototype is currently in the testing and pre-production phase. Testing is expected to complete by Q2/2023, and large-scale integration of the framework into the company's existing code bases is planned to commence shortly thereafter.

List of Figures

| | | |
|------|---|----|
| 4.1 | Enforcing type-bound configuration methods in EF Core models. | 12 |
| 4.2 | Using generics to enforce type-safety and encapsulation. | 13 |
| 4.3 | Physical separation of configuration from business logic using <code>partial</code> classes. | 14 |
| 4.4 | The TPH mapping strategy in RECAP. | 14 |
| 4.5 | The TPC mapping strategy in RECAP. | 15 |
| 4.6 | Proposed namespace and sub-library structure for stored procedure mapping. | 17 |
| 4.7 | Procedure configuration API implementation of a stored procedure targeting PL/SQL. | 18 |
| 4.8 | Proposed type structure for the <code>Compiler</code> namespace using a split build process. | 20 |
| 4.9 | Output types of the build process | 20 |
| 4.10 | The <code>ProcedureCache</code> is used to create concrete PCO instances of generic type <code>T</code> | 21 |
| A.1 | JIT and AOT in .NET 7 | 43 |
| A.2 | The structure of the Common Language Infrastructure (CLI) and the JIT compilation process | 44 |
| D.1 | Benchmark results for property accessor generation. | 62 |
| D.2 | Box plot for property accessor generation. | 63 |
| D.3 | Measurement inaccuracies in mean execution time of the direct getter for value types. | 64 |
| D.4 | Box plot displaying the measured performance overhead of RECAP compared to direct invocation. | 67 |

List of Tables

| | | |
|-----|---|----|
| 3.1 | Summary of requirements for RECAP. | 11 |
| 5.1 | Validation of requirements. | 25 |
| D.1 | Benchmark results (execution times and memory allocations). | 68 |

List of Listings

| | | |
|----|--|----|
| 1 | A combination of data annotations and convention-based mapping. | 7 |
| 2 | Example of an entity (left) configured employing a hybrid of data annotations in the entity class itself and fluent API configuration in the <code>DbContext</code> class (right). | 8 |
| 3 | Excerpt of the proposed configuration syntax for the example shown in figure 4.7. | 19 |
| 4 | A method adding two integers and returning the result | 45 |
| 5 | The IL representation of the <code>Math</code> class and its <code>Add</code> method | 45 |
| 6 | The native x86 machine code generated by the JIT compiler | 46 |
| 7 | A simple lambda function (left) and its decompiled representation (right) . | 49 |
| 8 | A simple lambda expression (left) and its generated expression tree representation (right) | 50 |
| 9 | <code>Customer</code> | 50 |
| 10 | Annotation-based entity configuration. | 51 |
| 11 | Annotation-based entity configuration with column name override. | 51 |
| 12 | Fluent API mapping of the <code>Customer</code> table | 51 |
| 13 | LINQ query to retrieve all customers with orders placed after 1/1/1997. . . | 52 |
| 14 | Mapping an entity to stored procedures for Insert, Update, Delete operations. | 53 |
| 15 | Mapping a stored procedure returning all columns of an entity as the result of the procedure. | 53 |
| 16 | Invoking a stored function using raw SQL and <code>DbParameter</code> objects. | 54 |
| 17 | Reflective enumeration of types implementing <code>IReflectiveModelConfiguration<T></code> . 55 | |
| 18 | Invocation of a reflective configuration method. | 56 |
| 19 | Reflectively loading and invoking base class configuration code. | 57 |
| 20 | Enforcing naming and mapping policies. | 57 |
| 21 | Late-bound property access using reflection. | 58 |
| 22 | Late-bound property access using reflection and compiler-generated backing fields. | 59 |
| 23 | Excerpt of early bound property access using expression trees. | 59 |
| 24 | Excerpt of raw IL byte code generation using the <code>Reflection.Emit</code> API. . | 60 |
| 25 | “ <i>Direct</i> ” early-bound property access. | 61 |
| 26 | Performance benchmark for stored procedure invocation using RECAP and raw SQL. | 66 |
| 27 | Stored procedure used for benchmarking. | 66 |
| 28 | Early bound property access using expressions. | 70 |

| | | |
|----|--|----|
| 29 | Micro-optimized IL byte code-based accessor generation using the <code>Reflection.Emit</code> API. | 73 |
|----|--|----|

Acronyms

AGU address generation unit [46](#)

ALU arithmetic logic unit [46](#)

AOT ahead-of-time [28](#), [43](#), [46](#)

API application programming interface [2](#), [3](#), [4](#), [5](#), [12](#), [17](#), [18](#), [21](#), [22](#), [24](#), [28](#), [30](#), [31](#), [34](#), [35](#), [45](#), [48](#), [50](#), [56](#), [60](#), [61](#), [63](#), [73](#)

BCL Base Class Library [47](#), [48](#)

CIL Common Intermediate Language [3](#), [43](#), [44](#), [46](#)

CLI Common Language Infrastructure [28](#), [33](#), [43](#), [44](#), [45](#), [47](#), [48](#)

CLR Common Language Runtime [3](#), [4](#), [10](#), [43](#), [45](#), [46](#), [47](#), [48](#)

CLS Common Language Specification [43](#), [44](#), [45](#), [47](#)

CoreCLR .NET Core Common Language Runtime [43](#), [48](#)

CRUD create, read, update, and delete [5](#), [10](#), [53](#)

CTS Common Type System [44](#), [45](#), [47](#), [48](#)

EF Core Entity Framework Core [ii](#), [1](#), [2](#), [3](#), [4](#), [5](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [14](#), [15](#), [16](#), [17](#), [18](#), [25](#), [28](#), [33](#), [34](#), [35](#), [39](#), [40](#), [43](#), [50](#), [51](#), [52](#), [53](#), [54](#), [59](#)

IL Intermediate Language [3](#), [4](#), [12](#), [24](#), [25](#), [27](#), [30](#), [31](#), [35](#), [43](#), [44](#), [45](#), [46](#), [47](#), [48](#), [49](#), [50](#), [60](#), [61](#), [62](#), [63](#), [73](#)

JIT just-in-time [3](#), [28](#), [30](#), [43](#), [44](#), [45](#), [46](#), [47](#)

LINQ Language Integrated Query [34](#)

ORM object-relational mapping [ii](#), [1](#), [2](#), [3](#), [4](#), [5](#), [7](#), [10](#), [11](#), [25](#), [26](#), [33](#), [35](#)

PCO procedure command object [17](#), [35](#)

RECAP Reflective Entity Configuration And Procedure mapping framework [ii](#), [1](#), [35](#)

RISC reduced instruction set computer [46](#)

TPC table-per-concrete-type [11](#), [14](#), [15](#), [23](#), [25](#), [26](#), [28](#), [56](#)

TPH table-per-hierarchy [11](#), [14](#), [15](#), [25](#), [28](#)

TPT table-per-type [11](#), [14](#), [15](#), [25](#), [26](#)

VES Virtual Execution System [34](#), [43](#), [47](#)

Glossary

ADO.NET ADO.NET is a set of libraries provided by Microsoft for accessing and manipulating data from a variety of sources, including databases, data feeds, and files. It provides classes such as `DbCommand` and `DataReader` for executing commands and reading data, as well as classes for defining connections and transactions. ADO.NET is often used in conjunction with [Entity Framework Core](#), a higher-level [ORM](#) library built on top of ADO.NET. [2](#), [10](#), [21](#), [22](#), [26](#), [34](#), [53](#), [54](#), [66](#), [68](#)

auto-property C# allows shortened syntax for property definitions, known as [auto-properties](#), where the compiler generates a getter, setter, and backing field for the property. The syntax is: `public int Foo { get; set; }` [11](#), [24](#), [27](#), [33](#), [58](#)

BenchmarkDotNet BenchmarkDotNet is a performance evaluation tool for .NET applications that aims to provide precise, consistent, and reproducible measurements of code performance through automation of benchmark execution and analysis. It employs various methods to eliminate sources of variability and ensure the validity of benchmark results as a reflection of the actual performance of the code under examination. This tool is frequently utilized by developers to optimize code performance and compare the efficiency of different algorithms or implementations [[NET22b](#)]. [61](#)

contravariant A property of generic interfaces and delegate types in C# where the type parameter is allowed to vary in the opposite direction as the interface or delegate. This means that if an interface or delegate is declared as `IFoo<in T>` (contravariant), a more general type can be used as the type argument than what is specified in the interface or delegate. [34](#)

covariant A property of generic interfaces and delegate types in C# where the type parameter is allowed to vary in the same direction as the interface or delegate. This means that if an interface or delegate is declared as `IFoo<out T>` (covariant), a more specific type can be used as the type argument than what is specified in the interface or delegate. [34](#)

delegate In C#, a delegate is a type-safe function pointer that can be used to represent a method or function. Delegates can be used to pass methods as arguments to other methods and can be dynamically invoked at runtime. [4](#), [20](#), [21](#), [22](#), [47](#), [49](#), [50](#), [60](#)

ECMA-335 The ECMA-335 [Common Language Infrastructure \(CLI\)](#) specification [[ECM12](#)]. [33](#), [43](#)

ECMA-334 The ECMA-334 C# language specification [[ECM22](#)]. [27](#), [47](#), [49](#), [59](#)

ECMA-335 CLI Specification Addendum The ECMA-335 CLI Specification Addendum is “a list of additions and edits to be made in [ECMA-335](#) specifications”. These

include additional features like `static abstract` interface members as well as fixes to specification errors [NET21b]. 12

execution context The `execution context` of a `represent` represents the context in which the procedure is executed for a given thread. The `execution context` manages anything state-related during an invocation, e.g. the `ADO.NET` database parameter objects used for a specific invocation. 21, 22, 34, 66, 67, 68

I/O container The Input/Output container is a procedure-specific object that is used to pass arguments to `RECAP` and retrieve the results of the stored procedure upon invocation. The `I/O container` is tightly coupled with the corresponding stored procedure and is configured in a similar manner to entities in `EF Core` using a fluent configuration syntax. 17, 18, 19, 20, 21, 22, 23, 24, 25, 34, 60, 61, 63, 68

init-only `Init-only` accessors are a feature introduced in C#9.0+ that allows for the creation of immutable properties in classes. They are defined using the `init` keyword and can only be assigned during instance initialization or in the constructor of the class. `Init-only` accessors are commonly used in `record` types, which are designed for simple data storage and minimized code redundancy for `init-only` property declaration. `RECAP` bypasses `init-only` write restrictions to allow developers to use the shortened record syntax for `I/O containers`. 11, 19, 24, 34

invariant In the context of generic interfaces in C#, invariance refers to the property that a generic interface is neither `covariant` nor `contravariant` with respect to its type parameters. This means that a generic interface with a type parameter `T` cannot be treated as a subtype of a generic interface with a type parameter `U`, even if `T` is a subtype of `U`. For example, the interface `IEnumerable<object>` cannot be used in place of `IEnumerable<T>`, even though every type `T` extends `object`. 15

LINQ `Language Integrated Query (LINQ)` is a C# language feature that provides a unified syntax for querying and manipulating data from various sources, including in-memory collections, databases, and XML documents. `LINQ` uses a fluent syntax, which allows developers to chain together query operations using method calls rather than writing complex SQL or XPath queries. 4, 5, 30, 39, 52, 55, 59

member access lambda expression A `member access lambda expression` in C# is a type of `expression tree` that projects an object onto one of its members. It is written as `object => object.Member` and can be used to retrieve metadata information about the accessed member via reflection. 4, 19, 20, 34

P/Invoke Platform Invoke (P/Invoke) is a mechanism provided by the .NET Framework that allows managed code (code that is executed by the `Virtual Execution System (VES)`) to call native code (code that is compiled to machine code) and vice versa. It is implemented through a set of `APIs` that provide access to the functions exported by

dynamic-link libraries (DLLs) and to functions in unmanaged code. [P/Invoke](#) is often used to access functionality provided by operating system [APIs](#) or by third-party libraries that are not available in the .NET Framework. [35](#), [45](#)

PCO The term [procedure command object \(PCO\)](#) refers to the singleton class or instance that represents a concrete stored database procedure or function in [RECAP](#). [PCOs](#) partially implement the command pattern, as the inclusion of an `Invoke()` method is merely a convention rather than a contract enforced through an implemented interface or another mechanism. Since [PCO](#) classes represent the interface between [RECAP](#) and client code, developers are given the option to choose the most suitable signature for one or more `Invoke()` methods based on the specific needs and goals of the procedure. [17](#), [18](#), [19](#), [21](#), [22](#), [25](#), [28](#), [34](#), [35](#), [68](#)

procedure configuration API The [procedure configuration API](#) defines the class structure that must be implemented by client code when using [RECAP](#) for stored procedure mapping. [17](#), [18](#), [28](#), [35](#)

RECAP [Reflective Entity Configuration And Procedure mapping framework \(RECAP\)](#) is the common name of the reflection-based [ORM](#) configuration and stored procedure mapping framework, which is proposed and implemented in the context of this work. Based on [Entity Framework Core \(EF Core\)](#), [RECAP](#) represents a technology-agnostic standardization of Fluent API model configurations, adding support for policies to enforce explicit naming and mapping conventions as well as providing a command-object pattern for stored procedure and function mapping in Oracle SQL and MySQL. [ii](#), [1](#), [2](#), [3](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [34](#), [35](#), [55](#), [56](#), [66](#), [67](#), [68](#)

Reflection API The `System.Reflection` namespace in .NET provides a set of classes and methods that allow a program to inspect and manipulate the metadata of assemblies and types at runtime. This allows for the dynamic enumeration of types and their members, as well as the ability to load and obtain information about the structure of an assembly. [3](#), [24](#), [45](#), [48](#), [50](#), [56](#)

Reflection.Emit API The `System.Reflection.Emit` namespace in .NET provides a set of classes and methods that allow a program to emit new [IL](#) instructions at runtime. This allows for the dynamic creation of new types, fields, methods, and properties, as well as the ability to define custom attributes. `System.Reflection.Emit` is a sub-namespace of `System.Reflection`. *see* [Reflection API](#), [4](#), [24](#), [30](#), [31](#), [48](#), [60](#), [61](#), [63](#), [73](#)

Roslyn [Roslyn](#) is the open-source compiler for the C# and Visual Basic .NET programming languages. It was introduced in 2014 as part of the .NET Compiler Platform and provides a set of [APIs](#) for parsing, analyzing, and modifying C# and VB code. [Roslyn](#) is used by Visual Studio and other tools for code analysis, refactoring, and other code-related tasks. [4](#), [24](#), [27](#), [35](#), [44](#), [59](#), [60](#)

SharpLab “*SharpLab is a .NET code playground that shows intermediate steps and results of code compilation. [...] SharpLab allows you to see the code as [the] compiler sees it, and get a better understanding of .NET languages.*” [Shc19] 41, 49

References

Literature

- [Che+16] Tse-Hsun Chen et al. “*Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks*”. In: *IEEE Transactions on Software Engineering* (2016), pp. 9–10 (cit. on p. 5).
- [ECM12] ECMA International. *ECMA-335: Common Language Infrastructure (CLI)*. Software engineering and interfaces ISO/IEC 23271. Rue du Rhone 114 CH-1204 Geneva, June 2012, pp. 9–12, 16, 18, 25, 39, 47–48, 53, 69–79, 122–127, 140, 182–186, 290, 438, 440. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf (visited on 12/28/2022) (cit. on pp. 3, 4, 33, 43–48).
- [ECM22] ECMA International. *ECMA-334: C# language specification*. Software engineering and interfaces ISO/IEC 23270. Rue du Rhone 114 CH-1204 Geneva, June 2022, pp. 83–84, 387–388. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-334_6th_edition_june_2022.pdf (visited on 12/28/2022) (cit. on pp. 4, 33, 48, 49, 59).
- [Fog22] Agner Fog. *4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Nov. 2022. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 01/08/2023) (cit. on p. 46).
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, pp. 184–194, 306–309, 322–324 (cit. on p. 5).
- [Gor22] Brian L. Gorman. *Practical Entity Framework Core 6: Database Access for Enterprise Applications*. 2nd Edition. New York, NY: Apress Media LLC, 2022, pp. 324–330 (cit. on p. 10).
- [Int22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual: Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Order Number: 325462-078US. Dec. 2022, pp. 3–594 Vol. 2A, 3–595 Vol. 2A. URL: <https://cdrdv2.intel.com/v1/dl/getContent/671200> (visited on 01/08/2023) (cit. on p. 46).
- [Kok18] Konrad Kokosa. *Pro .NET Memory Management. For better Code, Performance and Scalability*. Apress Media LLC, 2018, pp. 273, 321–322, 238, 846–848 (cit. on pp. 3, 45, 47).

- [SKS20] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th Edition. Previous editions published in 2011, 2006, and 2002. New York, NY: McGraw-Hill Education, 2020, pp. 381–382 (cit. on p. 5).

Online sources

- [EZ20] J. Evain and A. Zaytsev. *Mono.Reflection on GitHub: “Some useful reflection helpers, including an IL disassembler.”* 2020. URL: <https://github.com/jbevain/mono.reflection> (visited on 01/05/2023) (cit. on p. 27).
- [Ic23] IronLanguages and contributors. *IronPython & IronRuby – IronLanguages on GitHub*. 2023. URL: <https://github.com/IronLanguages> (visited on 01/08/2023) (cit. on p. 44).
- [Mic21] Microsoft Docs. *What is managed code?* 2021. URL: <https://learn.microsoft.com/en-us/dotnet/standard/managed-code> (visited on 12/28/2022) (cit. on pp. 3, 45).
- [Mic22] Microsoft Corporation. *Northwind sample database*. 2022. URL: <https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs> (visited on 12/23/2022) (cit. on p. 50).
- [NET20a] .NET-Foundation. *Entity Framework Core issue “FromSqlRaw() and ParameterDirection.Output” on GitHub*. 2020. URL: <https://github.com/dotnet/efcore/issues/21433> (visited on 01/08/2023) (cit. on p. 10).
- [NET20b] .NET-Foundation. *Managing Database Schemas – Entity Framework Core*. 2020. URL: <https://learn.microsoft.com/en-us/ef/core/managing-schemas/> (visited on 01/07/2023) (cit. on p. 5).
- [NET20c] .NET-Foundation. *Tables class on GitHub, part of the .NET Runtime*. 2020. URL: <https://github.com/dotnet/runtime/blob/bd175dae3a6fca2986db6135318add78867eceab/src/libraries/System.Reflection.Metadata/src/System.Reflection.Metadata/Internal/Tables.cs#L1461> (visited on 12/26/2022) (cit. on p. 58).
- [NET21a] .NET-Foundation. *Binder class used internally by the dynamic keyword in C# for late-binding on class members*. 2021. URL: <https://github.com/dotnet/runtime/blob/57bfe474518ab5b7cfe6bf7424a79ce3af9d6657/src/libraries/Microsoft.CSharp/src/Microsoft.CSharp/RuntimeBinder/Binder.cs> (visited on 12/06/2022) (cit. on p. 48).

- [NET21b] .NET-Foundation. *ECMA-335 CLI Specification Addendum*. 2021. URL: <https://github.com/dotnet/runtime/blob/ddb91f564af56f41720686e8e74659e35ae46475/docs/design/specs/Ecma-335-Augments.md#static-interface-methods> (visited on 12/06/2022) (cit. on pp. 13, 34).
- [NET21c] .NET-Foundation. *QueryCompiler class on GitHub, part of EF Core. Using Expression Trees to compile LINQ queries into SQL*. 2021. URL: <https://github.com/dotnet/efcore/blob/767bc93c6e6fca19868f486c7df8f4235bcf53cd/src/EFCore/Query/Internal/QueryCompiler.cs> (visited on 01/07/2023) (cit. on p. 4).
- [NET22a] .NET-Foundation. *BackingFieldConvention class used by EF Core to resolve backing fields by name*. 2022. URL: <https://github.com/dotnet/efcore/blob/767bc93c6e6fca19868f486c7df8f4235bcf53cd/src/EFCore/Metadata/Conventions/BackingFieldConvention.cs#L161> (visited on 12/26/2022) (cit. on p. 59).
- [NET22b] .NET-Foundation. *BenchmarkDotNet - Powerful .NET library for benchmarking*. 2022. URL: <https://benchmarkdotnet.org/> (visited on 12/23/2022) (cit. on p. 33).
- [NET22c] .NET-Foundation. *CollectionNavigationBuilder class on GitHub, part of EF Core. One of the places where member access expressions are used for mapping*. 2022. URL: <https://github.com/dotnet/efcore/blob/f7328757bb8120ed3284a6a136a40bebe5e891a3/src/EFCore/Metadata/Builders/CollectionNavigationBuilder%5C%60.cs#L74> (visited on 01/07/2023) (cit. on p. 4).
- [NET22d] .NET-Foundation. *Creating and Configuring a Model – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/modeling/> (visited on 01/07/2023) (cit. on p. 5).
- [NET22e] .NET-Foundation. *Database Providers – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/providers/> (visited on 01/07/2023) (cit. on p. 5).
- [NET22f] .NET-Foundation. *DbContext Class – Entity Framework Core API Reference*. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext> (visited on 01/07/2023) (cit. on p. 5).
- [NET22g] .NET-Foundation. *DbSet<T> class on GitHub, part of EF Core*. 2022. URL: <https://github.com/dotnet/efcore/blob/767bc93c6e6fca19868f486c7df8f4235bcf53cd/src/EFCore/DbSet.cs#L46> (visited on 01/07/2023) (cit. on p. 5).

- [NET22h] .NET-Foundation. *EcmaFormatRuntimePropertyInfo* class used to represent property metadata in the ECMA-335 format. 2022. URL: <https://github.com/dotnet/runtime/blob/b15ffe57b9a1c4e8aaa2f49125efeace697fa9fb/src/coreclr/nativeaot/System.Private.CoreLib/src/System/Reflection/Runtime/PropertyInfos/EcmaFormat/EcmaFormatRuntimePropertyInfo.cs> (visited on 12/26/2022) (cit. on p. 58).
- [NET22i] .NET-Foundation. *Entity Framework Core* issue “Implement stored procedure update mapping” on GitHub. 2022. URL: <https://github.com/dotnet/efcore/pull/28553> (visited on 01/08/2023) (cit. on p. 10).
- [NET22j] .NET-Foundation. *Entity Types – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/modeling/entity-types> (visited on 01/07/2023) (cit. on p. 5).
- [NET22k] .NET-Foundation. *ExpandoObject Class – .NET API Reference*. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.dynamic.expandoobject?view=net-7.0> (visited on 01/08/2023) (cit. on p. 48).
- [NET22l] .NET-Foundation. *ExpressionExtensions* class on GitHub, part of *EF Core*. Using Expression Trees to retrieve *MemberInfo* metadata. 2022. URL: <https://github.com/dotnet/efcore/blob/3a62379ec775eb8e6a6ef7334212005b80faadb7/src/EFCore/Infrastructure/ExpressionExtensions.cs> (visited on 01/07/2023) (cit. on p. 4).
- [NET22m] .NET-Foundation. *Inheritance – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/modeling/inheritance> (visited on 01/08/2023) (cit. on pp. 14, 15).
- [NET22n] .NET-Foundation. *MetadataReader* class used to read IL metadata on runtime. 2022. URL: <https://github.com/dotnet/runtime/blob/bd175dae3a6fca2986db6135318add78867eceab/src/libraries/System.Reflection.Metadata/src/System/Reflection/Metadata/MetadataReader.cs#L1240> (visited on 12/26/2022) (cit. on p. 58).
- [NET22o] .NET-Foundation. *Native AOT Deployment*. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/> (visited on 01/07/2023) (cit. on p. 43).
- [NET22p] .NET-Foundation. *Performance – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/performance/> (visited on 01/07/2023) (cit. on p. 5).

- [NET22q] .NET-Foundation. *PropertyDefinition class on GitHub, part of the .NET Runtime*. 2022. URL: <https://github.com/dotnet/runtime/blob/bd175dae3a6fca2986db6135318add78867eceab/src/libraries/System.Reflection.Metadata/src/System/Reflection/Metadata/TypeSystem/PropertyDefinition.cs> (visited on 12/26/2022) (cit. on p. 58).
- [NET22r] .NET-Foundation. *RelationalQueryableExtensions.FromSql() – Entity Framework Core API Reference*. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.relationalqueryableextensions.fromsql?view=efcore-7.0> (visited on 01/07/2023) (cit. on p. 10).
- [NET22s] .NET-Foundation. *RuntimePropertyInfo class used to represent property metadata*. 2022. URL: <https://github.com/dotnet/runtime/blob/b15ffe57b9a1c4e8aaa2f49125efeace697fa9fb/src/coreclr/nativeaot/System.Private.CoreLib/src/System/Reflection/Runtime/PropertyInfos/RuntimePropertyInfo.cs> (visited on 12/26/2022) (cit. on p. 58).
- [NET22t] .NET-Foundation. *RyuJIT Compiler Structure*. 2022. URL: <https://github.com/dotnet/runtime/blob/e4ada8481aba5a7515ddce704dac15ac95cadcbd/docs/design/coreclr/jit/ryujit-overview.md> (visited on 01/08/2023) (cit. on p. 43).
- [NET22u] .NET-Foundation. *System.Reflection.Emit Namespace – .NET API Reference*. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit?view=net-7.0> (visited on 01/08/2023) (cit. on p. 48).
- [NET22v] .NET-Foundation. *User-defined function mapping – Entity Framework Core*. 2022. URL: <https://learn.microsoft.com/en-us/ef/core/querying/user-defined-function-mapping> (visited on 01/07/2023) (cit. on p. 10).
- [NET23] .NET-Foundation. *Entity Framework Core on GitHub*. 2023. URL: <https://github.com/dotnet/efcore> (visited on 01/07/2023) (cit. on p. 5).
- [Shc19] Andrey Shchekin. *SharpLab README on GitHub*. 2019. URL: <https://github.com/ashmind/SharpLab/blob/c9a0df24510bbe3123e3e70126d82777ea754ef8/README.md> (visited on 12/29/2022) (cit. on p. 36).

Appendix contents

| | | |
|-------|---|----|
| A | Extended theoretical foundation | 43 |
| A.1 | An introduction to .NET | 43 |
| A.1.1 | .NET Common Language Infrastructure | 43 |
| A.1.2 | The Common Language Specification | 44 |
| A.1.3 | The Common Intermediate Language and the metadata | 44 |
| A.1.4 | The Virtual Execution System | 47 |
| A.1.5 | The Common Type System | 47 |
| A.1.6 | Reflection | 48 |
| A.1.7 | Expression trees | 49 |
| A.2 | Entity Framework Core examples | 50 |
| B | Extended problem analysis | 53 |
| B.1 | Current practices for procedure mapping | 53 |
| B.1.1 | CRUD operations | 53 |
| B.1.2 | Raw SQL | 54 |
| C | Extended implementation | 55 |
| C.1 | Model builder injection | 55 |
| C.1.1 | Reflective type enumeration | 55 |
| C.1.2 | Reflective configuration | 56 |
| C.1.3 | Table per concrete type support | 56 |
| C.1.4 | Policy enforcement | 57 |
| C.2 | Iterative accessor optimization | 58 |
| C.2.1 | Late-bound property access | 58 |
| C.2.2 | Expressions | 59 |
| C.2.3 | Reflection emit | 60 |
| D | Performance benchmarks | 61 |
| D.1 | Methodology | 61 |
| D.2 | Property accessor optimization | 61 |
| D.3 | Overall performance evaluation | 65 |
| E | Full implementations | 69 |
| E.1 | Expression tree-based accessors | 69 |
| E.2 | IL emitted accessors | 70 |

A Extended theoretical foundation

This appendix chapter provides an in-depth introduction to the .NET platform and [EF Core](#), including numerous examples and additional background information.

A.1 An introduction to .NET

.NET is a platform for building and running applications in a managed virtual machine, making it operating system- and hardware-independent. Additionally, it is designed to be language-independent, allowing a multitude of programming languages to be used (e.g., C#, VB.NET, and F#). In order to enable a single .NET application to run on different platforms, source code is first compiled into byte code, referred to as [Common Intermediate Language](#) (CIL or IL), which can then be distributed and executed on any platform with a .NET runtime installed [[ECM12](#), pp. 9–12, 290].

The .NET runtime is officially referred to as the [.NET Core Common Language Runtime](#) ([CoreCLR](#) or [CLR](#))⁶ and is responsible for compiling IL into the native machine code of the target platform, managing memory, and providing a set of services to the application.

Figure A.1 illustrates this type of compilation, which is referred to as [just-in-time \(JIT\)](#) compilation, as it is performed right before the code is executed. The [JIT](#) compiler uses a tiered compilation approach where the code is first compiled into a low-quality version of the native machine code and then optimized over time as the code is executed more often. Optimizations performed by the [JIT](#) Compiler include inlining, Flowgraph Analysis (e.g., loop unrolling), and Loop Invariant Code Hoisting (i.e., moving code out of loops if possible). Starting with the latest version (.NET 7),

support for [ahead-of-time \(AOT\)](#) compilation has been added, allowing IL to be compiled into platform- and architecture-specific native machine code as part of the build process. [AOT](#) compilation allows for the distribution of native machine code without the need for a .NET runtime to be installed on the target system or to be embedded into the application. While this allows for better performance and smaller file sizes, it also (naturally) strips out any type metadata of the IL code, making it impossible to apply dynamic approaches like reflection [[ECM12](#), pp. 72–79, 184–186][[NET22t](#)][[NET22o](#)].

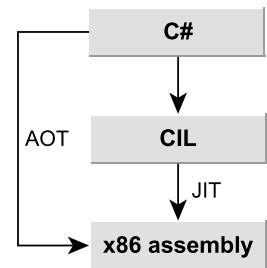


Figure A.1: [JIT](#) and [AOT](#) in .NET 7

A.1.1 .NET Common Language Infrastructure

The [Common Language Infrastructure \(CLI\)](#) specification [ECMA-335](#) defines the core aspects of .NET. These include the [Common Language Specification \(CLS\)](#), the [Virtual](#)

⁶ In this work, [CLR](#) and [CoreCLR](#) are used interchangeably because the differences in platform support between the two are not of relevance in this context.

Execution System (VES), the Common Type System (CTS), and the metadata format needed for reflection and for interoperability between these [ECM12, p. 9].

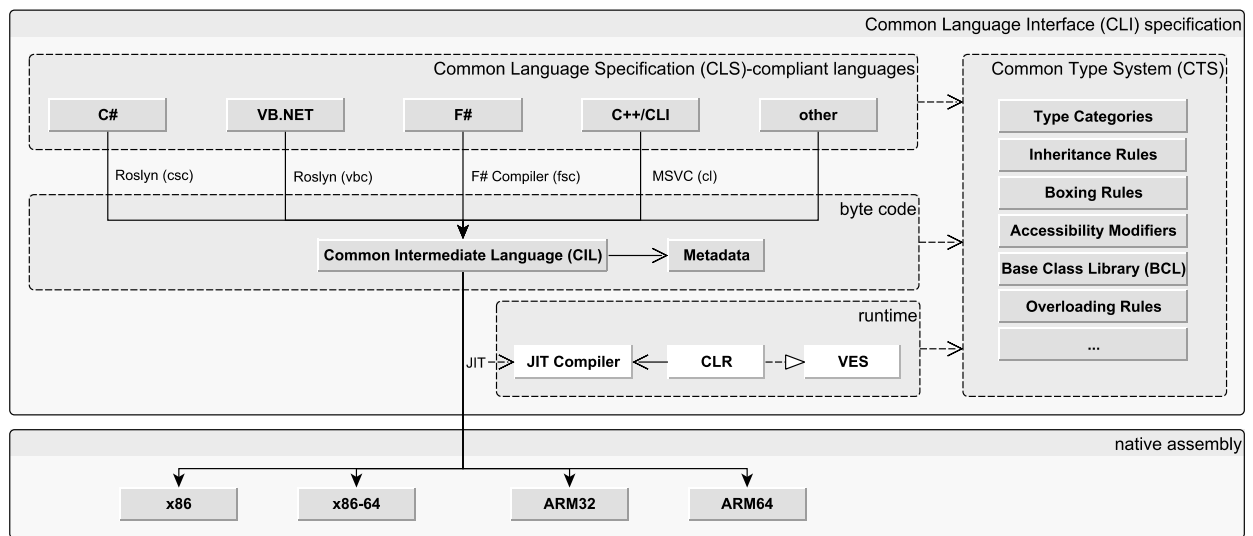


Figure A.2: The structure of the CLI and the JIT compilation process

Figure A.2 illustrates how the different components of the CLI specification interact with each other during the JIT compilation process. The following sections will describe these different components in more detail.

A.1.2 The Common Language Specification

The Common Language Specification (CLS) allows for interoperability between different programming languages. As such, it defines a set of rules (e.g., calling conventions, the type system, ...) that all programming languages targeting the CLI must follow. This standardization allows, for example, a library written in C++/CLI to be loaded and used by a C# application as both languages follow the same rules and are compiled into the same IL format [ECM12, pp. 69–71].

The similarity in the language design even allows different languages to use the same shared compiler and code analyzer platform, e.g., Roslyn for C# and VB.NET, with different language-specific compiler implementations, csc (CSharp Compiler) and vbc (Visual Basic Compiler) respectively. Furthermore, other CLI languages like F#, C++/CLI, or even unsupported languages (e.g., IronPython, IronRuby, or Swift) may use their own Common Language Specification (CLS) compliant compiler implementations to ensure interoperability with .NET [Ic23].

A.1.3 The Common Intermediate Language and the metadata

The Common Intermediate Language (CIL) is the byte code format used by the CLI. IL is a stack-based virtual instruction set that is compiled from the source code of a .NET

application. At runtime, the **IL** code is then compiled into native machine code by the **JIT** compiler. In addition to being platform-independent, **IL** contains type metadata used by the **JIT** compiler for optimization and by the **CLR** for performing type checks, boxing, and type casting. This metadata includes information about the types and members of the application, including their names, visibility, and attributes. These metadata annotations are used by the **CLR** and **Common Type System (CTS)** to provide the **Reflection API**, which allows for the dynamic inspection of the application's types and members based on their metadata. The metadata also contains the required information needed for interoperability with other **CLI** languages. This includes member layout information for the types, allowing for the correct marshaling of data between different languages, **P/Invoke** signatures for interoperability with native code, and information about the **CLS** compliance of the types and members [Kok18, p. 238][Mic21][ECM12, pp. 53, 122–127]. The following example aims to illustrate the structure of **IL** code and how it is used to represent a simple C# method.

```
public static class Math
{
    public static int Add(int a, int b) => a + b;
}
```

Listing 4: A method adding two integers and returning the result

Listing 4 features a **class** called **Math** with **public** visibility, meaning it can be accessed from outside the assembly. The class contains a **public** method called **Add** that takes two integers as parameters and returns the sum of the two. Both the class and the method are marked as **static**, meaning they can be accessed without needing to instantiate the class. The resulting **IL** byte code of the **Math** class and its **Add** method is shown in listing 5.

```
1 // class, visibility, layout, marshaling info, abstract sealed = static, class name
2 .class public auto ansi abstract sealed beforefieldinit Math
3     extends [System.Runtime]System.Object // inheritance metadata
4 {
5     // Methods
6     .method public hidebysig static // method, visibility, info for tooling, static
7         int32 Add (                // method signature
8             int32 a,
9             int32 b
10          ) cil managed              // standard IL code, managed by CLR
11 { // Method begins at RVA 0x2067
12     // Code size 4 (0x4)
13     .maxstack 8                    // stack size
14     IL_0000: ldarg.0                // load argument 0 (a)
15     IL_0001: ldarg.1                // load argument 1 (b)
16     IL_0002: add                    // add them
17     IL_0003: ret                    // return result
18 } // end of method Math::Add
19 } // end of class Math
```

Listing 5: The **IL** representation of the **Math** class and its **Add** method

Every expression beginning with a dot (.) is a metadata directive in IL. The first `.class` metadata directive defines the class and its `public` visibility, automatic class member layout (memory layout of the class members), hints to marshal strings in ANSI format, and that the class is both, `abstract` and `sealed`, meaning that it can neither be instantiated nor inherited, therefore implementing the `static` keyword from C#. The class is also marked as `beforefieldinit`, meaning that any static fields will be initialized before the class is used. Finally, the class name is stored [ECM12, p. 140].

Line 2 shows the inheritance metadata of the class, which is `System.Object` in this case. If class `Math` could be instantiated, the CLR would use this information to know that it is a reference type and thus must be allocated on the heap and can be implicitly cast to `System.Object`.

The following `.method` directive defines the `public static` `Add` method, with its full signature, and informs the CLR that the method contains managed CIL code (i.e., code that requires jitting) [ECM12, pp. 182–184].

The `.maxstack` directive specifies the maximum number of items that can be stored on the stack at any given time, which is 8 in this case. Finally, the `IL_0000` to `IL_0003` lines contain the executable IL instructions that can be jitted into native machine code.

```
Math.Add(Int32, Int32)
    L0000: lea eax, [ecx+edx]
    L0003: ret
```

Listing 6: The native x86 machine code generated by the JIT compiler

Listing 6 shows the native machine code generated by the JIT compiler for the `Add` method. The `lea` (load effective address) instruction loads the sum of the two arguments into the `eax` register, and the `ret` (return) instruction returns the value stored in the `eax` register. As can be seen, after jitting the IL code into native x86 machine code, all metadata information is lost, which is also why AOT-compiled code does not support reflection.

It is worth noting that the JIT compiler is able to optimize the IL code by replacing the `add` with the simpler `lea` instruction. This is done because the JIT compiler is aware that the carry flag is not used in this case. Thus the simpler `lea` instruction can be used, which evaluates the addition on the address generation unit (AGU), a distinct hardware used for address calculation, during the instruction decoding phase of the reduced instruction set computer (RISC) pipeline. This alleviates pressure on the arithmetic logic unit (ALU) during the execution phase of the RISC pipeline, allowing it to be used for other instructions executing simultaneously [Int22, 3-594 Vol. 2A, 3-595 Vol. 2A][Fog22].

A.1.4 The Virtual Execution System

The [Virtual Execution System \(VES\)](#) is the runtime environment described by the [CLI](#) specification, with the [CLR](#) being its implementation. The [VES](#) specifies all aspects of the runtime environment, including the aforementioned [JIT](#) compiler as well as the garbage collector, the threading model, and the security model [[ECM12](#), p. 72].

A.1.5 The Common Type System

While the [VES](#) is responsible for executing managed code and providing runtime services, the [Common Type System \(CTS\)](#) defines a portable type system that is supported by all other [CLI](#) components.

The [CTS](#) defines a standardized representation of types and their members in memory, enabling interoperability among various [CLS](#)-compliant languages as the underlying base-data types are shared. It is a rich type system that supports both static and dynamic typing. While static typing allows for type safety and early error detection during compile-time, resulting in better performance and more efficient code, dynamic typing allows for late binding and code generation at runtime [[ECM12](#), pp. 16, 25, 39, 290].

The [CTS](#) defines two main categories of static types: value types and reference types, with pointer types being a subset of the latter. Value types are stored on the stack and include primitive data types, such as integers or floating point numbers, and more complex types, such as enumerations, structures, and unions. Reference types are stored on the heap and include classes, interfaces, [delegates](#), pointers, or any boxed value type. Pointer types are further divided into unmanaged or native pointers and managed or byref pointers. Most of these types (except unmanaged function pointers) additionally support generic types, allowing for the definition of type parameters that can be used for type-safe code reuse [[ECM12](#), p. 18][[Kok18](#), pp. 273, 321–322, 846–848].

Dynamic typing can be achieved in .NET by casting class instances to `System.Object` or `System.ValueType` first before being cast to the desired type. This defers type evaluation until runtime, allowing late binding. Moreover, the [CLI](#) heavily relies on type metadata as dynamic dispatch via virtual method tables is used to achieve polymorphism for instance-based method invocation. The [Base Class Library \(BCL\)](#) also provides pseudo-dynamic concepts such as `ExpandoObject`, which uses a dictionary-like key-value data structure to mimic the creation and removal of properties at runtime but does not truly behave like a dynamic type as neither reflection nor type casting is supported. The C# language specification [ECMA-334](#) expands on the [CTS](#) by defining types with truly dynamic behavior. For example, in C#, the `dynamic` keyword can be used to defer evaluation of any type until runtime. This also allows for late binding on dynamically generated or loaded members, types, or assemblies that were not present on compile time due to usage of [IL](#) code emission or reflection. As this form of per-member dynamic dispatch is not natively supported by the

[CTS](#), the C# compiler generates a call to the `Microsoft.CSharp.RuntimeBinder.Binder` class, which is part of the [.NET Core Common Language Runtime \(CoreCLR\)](#) and implements a dynamic dispatch mechanism that is not part of the [CLI](#). Internally, the `Binder` class uses reflection on the [CLI](#)-compliant metadata to retrieve the correct member to be invoked [[ECM12](#), pp. 25, 47–48][[NET22k](#)][[ECM22](#), p. 84][[NET21a](#)].

A.1.6 Reflection

Reflection is the ability to inspect the metadata of an application at runtime, allowing for the dynamic discovery of types and members, their attributes, and their values.

The [CLR](#) provides the [Reflection API](#) based on the [CTS](#) and the metadata stored in the [IL](#) code. This [API](#) is exposed to the application via the `System.Reflection` namespace. It permits for dynamic loading of assemblies, the discovery of types and members, and the invocation of methods [[ECM12](#), pp. 438, 440].

The [BCL](#) additionally provides the [Reflection.Emit API](#) for the dynamic emission of [IL](#) code and the generation of new types and members at runtime. This can be used to dynamically emit [IL](#) op-codes to implement logic that is not known at compile-time, e.g., to generate a proxy class for a remote object dynamically or to explore and map a database table to a type [[NET22u](#)].

A.1.7 Expression trees

The [ECMA-334](#) (C#) specification introduces the capability to compile lambda expressions into [expression trees](#). An [expression tree](#) is a data structure that represents code in the form of a tree with nodes representing language constructs such as method calls or operators [[ECM22](#), pp. 83–84].

The following example uses [SharpLab](#) to demonstrate how a C# lambda function can be either compiled to [IL](#) or an [expression tree](#).

```
public class C
{
    public int I { get; set; }

    public static int M(C instance)
    {
        Func<C, int> lambda = c => c.I;
        return lambda(instance);
    }
}

public class C
{
    [Serializable]
    [CompilerGenerated]
    private sealed class <>c
    {
        public static readonly <>c <>9 = new <>c();

        [System.Runtime.CompilerServices.Nullable(new byte[] { 0,
            ↪ 1 })]
        public static Func<C, int> <>9__4_0;

        [System.Runtime.CompilerServices.NullableContext(1)]
        internal int <M>b__4_0(C c)
        {
            return c.I;
        }
    }
    ... // declaration of member "I" omitted for brevity
    [System.Runtime.CompilerServices.NullableContext(1)]
    public static int M(C instance)
    {
        return (<>c.<>9__4_0
            ?? (<>c.<>9__4_0 =
                new Func<C, int>(<>c.<>9.<M>b__4_0)))(instance);
    }
}
```

Listing 7: A simple lambda function (left) and its decompiled representation (right)

Listing 7 shows a simple lambda function and its compiled representation. The compiler lowers the lambda function to a static method in the generated nested class `<>c`. The compiler also generates a [delegate](#) that points to the newly created method, which is then lazily assigned and invoked.

The same code can be compiled into an [expression tree](#) by wrapping the lambda in an `Expression` type.

```

public static int M(C instance)
{
    Expression<Func<C, int>> lambda =
        c => c.I;
    return lambda.Compile()(instance);
}

[System.Runtime.CompilerServices.NullableContext(1)]
public static int M(C instance)
{
    ParameterExpression p = Expression.Parameter(typeof(C), "c");
    MemberExpression body = Expression.Property(p,
        (MethodInfo)MethodBase
            .GetMethodFromHandle((RuntimeMethodHandle));
    ParameterExpression[] array = new ParameterExpression[1];
    array[0] = p;
    return Expression.Lambda<Func<C, int>>(body, array)
        .Compile()(instance);
}

```

Listing 8: A simple lambda expression (left) and its generated [expression tree](#) representation (right)

Listing 8 shows how the same lambda is compiled into an [expression tree](#). The compiler translates the expression into a set of `Expression` objects representing the different parts of the lambda. The resulting [expression tree](#) can then be compiled into an executable [delegate](#) or dynamically inspected and modified at runtime. [Expression trees](#) expose the implementation of the lambda to the [Reflection API](#), while, by default, a lambda would be compiled into [IL](#), making runtime inspection difficult⁷.

A.2 Entity Framework Core examples

The following example illustrates how [Entity Framework Core \(EF Core\)](#) can be used in C# to retrieve data from a database and map it to a domain model. In this example, the Northwind sample database included with SQL Server will be used [Mic22].

```

public class Customer // Domain model class representing a customer in the Northwind database.
{
    public string CustomerID { get; set; } // Primary key of the customer table.
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    ... // Other properties omitted for brevity.
    // Navigation property for orders associated with this customer.
    public virtual ICollection<Order> Orders { get; } = new List<Order>();
}

```

Listing 9: Customer

Listing 9 shows an entity class representing a customer in the Northwind database. The `CustomerID`, `CompanyName`, `ContactName`, and `ContactTitle` properties correspond to columns in the customer table, while the navigation property `Orders` represents all orders associated with this customer. These navigation properties are automatically joined to the primary and foreign keys of the database tables.

[EF Core](#) uses conventions to determine how each property maps to a column in the database; however, these conventions can be overridden using attributes (annotations) or

⁷ It is possible to inspect the [IL](#) code of a lambda function, but this requires manually parsing the [IL](#) op-codes which may result in code that is challenging to maintain.

by modifying configuration using Fluent API. For example, default mapping for primary keys could be overridden by adding an attribute as follows:

```
public class Customer
{
    // Explicit primary key of the customer table (annotation overrides convention).
    [Key()]
    public string CustomerID { get; set; }
    ... // Other properties omitted for brevity.
}
```

Listing 10: Annotation-based entity configuration.

The `[Key]` attribute in listing 10 tells EF Core that the `CustomerID` property is a primary key and should be used as such when mapping to the database schema. In addition, the default column name could be overridden by specifying an additional parameter to the attribute, as shown in listing 11.

```
public class Customer
{
    [Key("Id")] // Override column name from "CustomerID" to "Id".
    public string CustomerID { get; set; } // Primary key of the customer table.
    ... // Other properties omitted for brevity.
}
```

Listing 11: Annotation-based entity configuration with column name override.

EF Core also supports fluent API configuration to override conventions and specify additional mapping information. For example, the `CustomerID` property could be configured as follows:

```
// Configure Customer entity using fluent API.
modelBuilder.Entity<Customer>(entity =>
{
    // use member access expression to select primary key property in domain model
    entity.HasKey(e => e.CustomerID);
    // Specify column name in database schema as "CustId".
    // override default convention-based mapping.
    entity.Property(e => e.CustomerID)
        .HasColumnName("Id");
    ... // Other configurations omitted for brevity...
    // Configure foreign key relationship of navigation property
    // for orders associated with this customer.
    entity.HasMany(e => e.Orders) 1 customer has n orders
        .WithOne(o => o.Customer) 1 order has 1 customer
        .HasForeignKey("CustomerID"); order has CustomerID FK
});
```

Listing 12: Fluent API mapping of the Customer table

Listing 12 demonstrates how a domain model class can be configured using the fluent API. In this case, the `Customer` entity type is to be mapped to the `Customers` table. The primary key of this table is set to be the `CustomerID` property and mapped to a column called `Id`. In

addition, the foreign key relationship between the `Customer` entity and its associated `Orders` is configured. These more complex configurations are only possible using the fluent API.

```
// Create new DbContext for Northwind database (usually injected with DI in ASP.NET).
var context = new NorthwindDbContext();
// Retrieve all customers with orders placed after 1/1/1997.
var customers = context.DbSet<Customers>() // Get Customers collection from DbSet<T>.
    .Where(customer => customer.Orders // Filter by Orders where...
        .Any(order => order.OrderDate > DateTimeOffset.Parse("1/1/1997"))); // ...any order date is after
    ↪ 1/1/1997.
    .ToList(); // translate the LINQ expression to SQL, execute it against the DB and return results as
    ↪ list.
```

Listing 13: LINQ query to retrieve all customers with orders placed after 1/1/1997.

Listing 13 shows how EF Core can be used in C# to retrieve data from a database and map it back into objects.

B Extended problem analysis

This appendix chapter thoroughly investigates procedure mapping approaches that are commonly used in conjunction with [EF Core](#) and [ADO.NET](#).

B.1 Current practices for procedure mapping

In order to provide a more detailed overview of existing implementation strategies, a brief analysis of the most prevalent approaches to stored procedure invocation used with [EF Core](#) is provided below.

B.1.1 CRUD operations

Invoking stored procedures for simple [CRUD](#) operations on entities is supported by [EF Core](#). This allows for mapping a procedure to an entity and invoking it using existing [fluent API](#)-based mapping. However, the procedure itself must be very basic and linked to a mapped entity. An example is shown in listing 14.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s => s
        .Update(u => u.HasName("modify_blog"))
        .Delete(d => d.HasName("delete_blog"))
        .Insert(i => i.HasName("insert_blog")));
```

Listing 14: Mapping an entity to stored procedures for Insert, Update, Delete operations.

Similarly, the result of a stored procedure can be mapped to an entity, but this requires the procedure to return all properties of the entity. This approach is shown in listing 15.

```
modelBuilder.Entity<MyEntity>().ToFunction("ReadMyEntity");
```

Listing 15: Mapping a stored procedure returning all columns of an entity as the result of the procedure.

These two approaches are supported by [EF Core](#) and can be used to invoke simple [CRUD](#) procedures. However, they do not provide any support for more complex scenarios where parameters other than those part of the mapped entity are used or when return values are required that are not part of this entity. Furthermore, it is not possible to map a function or stored procedure with [INOUT](#) or [OUT](#) return values in this way. Therefore, the use of raw SQL is necessary in these instances.

B.1.2 Raw SQL

Using EF Core's underlying ADO.NET DatabaseFacade presents a more flexible approach. Raw SQL queries can be used in conjunction with DbParameter objects for passing parameters and returning values. This approach is shown in listing 16. As can be seen, a lot of boilerplate code is required, which has to be written by hand each time a procedure or function needs to be invoked.

```
public string? GetReportHtmlOutput(int reportId, int? elementId, PrintInfoModel printInfo, string
↪ elementUrlPrefix)
{
    using OracleParameter fpnReportId = new("fpnReportId", OracleDbType.Int32, ParameterDirection.Input)
    {
        Value = reportId
    };
    using OracleParameter fpnReportElementId = new("fpnReportElementId", OracleDbType.Int32,
↪ ParameterDirection.Input)
    {
        Value = elementId
    };
    using OracleParameter fpsLanguage = new("fpsLanguage", OracleDbType.Varchar2, ParameterDirection.Input)
    {
        Value = string.IsNullOrEmpty(printInfo.TargetLanguage) ? Language.DE : printInfo.TargetLanguage
    };
    using OracleParameter fpsElementUrlPrefix = new("fpsElementUrlPrefix", OracleDbType.Varchar2,
↪ ParameterDirection.Input)
    {
        Value = elementUrlPrefix
    };
    using OracleParameter fpnCorrectionMode = new("fpnCorrectionMode", OracleDbType.Int32,
↪ ParameterDirection.Input)
    {
        Value = printInfo.CorrectionMode ? 1 : 0
    };
    using OracleParameter fpcrHtmlOutput = new("fpcrHtmlOutput", OracleDbType.Clob,
↪ ParameterDirection.Output);
    Database.ExecuteSqlRaw("BEGIN GETREPORTHTMLOUTPUT (:fpnReportId, :fpnReportElementId, :fpsLanguage, "
        + ":fpsElementUrlPrefix, :fpnCorrectionMode, :fpcrHtmlOutput); END;",
        fpnReportId, fpnReportElementId, fpsLanguage, fpsElementUrlPrefix, fpnCorrectionMode,
        ↪ fpcrHtmlOutput);
    return fpcrHtmlOutput.Value is OracleClob clob ? clob.Value : null;
}
```

Listing 16: Invoking a stored function using raw SQL and DbParameter objects.

This approach allows for the invocation of almost any stored procedure or function with support for various parameter types. At the same time, it can decrease the readability and maintainability of the code and requires knowledge of the syntax for stored procedure invocation specific to the underlying database engine, leading to reduced portability.

C Extended implementation

This appendix chapter provides additional implementation details for the features implemented in this work.

C.1 Model builder injection

C.1.1 Reflective type enumeration

During startup, the `LoadReflectiveModels()` extension method is invoked on the `ModelBuilder` instance to dynamically load all direct implementations of `IReflectiveModelConfiguration<T>` defined in the current application domain.

```
IEnumerable<ReflectiveEntity> entities = AssembliesWithEntryPoint()
    // get all types in these assemblies
    .SelectMany(asm => asm.GetTypes()
        .Where(type =>
            // only keep classes
            type.IsClass
            // only keep classes that implement IReflectiveModelConfiguration<T> where T is that class
            && type.ImplementsDirectGenericInterfaceWithTypeParameter(
                typeof(IReflectiveModelConfiguration<>), type)
            // only keep classes that have the specified database engine attribute if enabled
            && (databaseEngineAttributeType is null
                || type.GetCustomAttribute(databaseEngineAttributeType) is not null)))
    .Select(type => new ReflectiveEntity // collect results
    (
        Type: type,
        // get the exact Configure method declared by IModelConfiguration<T>
        Configure: type.GetMethod
        (// method name == "Configure" and method is public static, has EntityTypeBuilder<"type"> param
            nameof(EntityConfigInfoForReflection_DontChange.Configure), // don't hard-code "Configure"
            BindingFlags.Static | BindingFlags.Public | BindingFlags.DeclaredOnly,
            new Type[] { typeof(EntityTypeBuilder<>).MakeGenericType(type) })
        ))
    .Where(entity => entity.Configure is not null) // final cleanup of invalid results
    .ToArray();
```

Listing 17: Reflective enumeration of types implementing `IReflectiveModelConfiguration<T>`.

The [LINQ](#) expression in listing 17 is used to identify all entity types that should be loaded dynamically by [RECAP](#) in the current application domain. Among the types declared in assemblies containing the application entry point, only those implementing `IReflectiveModelConfiguration<T>` with `T` equal to the type of the implementing class are retained. To support the use of multiple database engines within a single application, a database engine-specific attribute can be specified to limit the enumeration to types marked with this attribute. This allows multiple `DbContext` classes to be defined and utilized in the same application, each configured for a different database engine. The final set of reflective entities is generated by merging the enumerated model types with the metadata for their corresponding `Configure()` method.

One of the main challenges encountered during the implementation of this feature was the absence of a built-in mechanism for filtering types based on their generic type parameters in the [Reflection API](#). To address this issue, [RECAP](#) employs its own set of extension methods to verify and enforce generic constraints on types and methods. For example, the `ImplementsDirectGenericInterfaceWithTypeParameter()` extension method shown in [listing 17](#) on the preceding page enforces the constraint that the implementing type must be identical to the generic type parameter of `IReflectiveModelConfiguration<T>`. The implementation of the stored procedure enumeration follows a similar approach but is more complex due to the increased usage of generic type constraints.

C.1.2 Reflective configuration

[Listing 18](#) shows the invocation of the `Configure()` method. First, the corresponding `EntityTypeBuilder<TEntity>` is constructed by calling the `Entity<T>()` factory method of the model builder with a generic type parameter that matches the entity type. The static `Configure()` method is then called on the entity class, passing the `EntityTypeBuilder<TEntity>` instance as an argument.

```
// get the generic Entity<T>() method
MethodInfo? entityTypeBuilderFactory = typeof(ModelBuilder).GetMethod(nameof(ModelBuilder.Entity), 1,
↳ Array.Empty<Type>());
// make it generic for our specific entity type
MethodInfo genericEntityTypeBuilderFactory = entityTypeBuilderFactory!.MakeGenericMethod(entity.Type);
// invoke it to create an EntityTypeBuilder<T> where T matches the entity
object entityTypeBuilderObj = genericEntityTypeBuilderFactory.Invoke(builder, null!);
parameters[0] = entityTypeBuilderObj!;
// invoke the Configure method with the EntityTypeBuilder<T> instance
entity.Configure!.Invoke(null, parameters);
```

Listing 18: Invocation of a reflective configuration method.

C.1.3 Table per concrete type support

[RECAP](#) supports the use of shared base class configuration code for [TPC](#) hierarchies by iterating up an entity's inheritance chain after its initial reflective configuration has been completed and then applying any shared configurations to that child entity.

```

1  Type? baseType = entity.Type.BaseType;
2  while (baseType is not null)
3  {
4      // iterate up the inheritance tree and look for any base class that implements
5      // IReflectiveBaseModelConfiguration<T> where T is the base class
6      if (baseType.ImplementsDirectGenericInterfaceWithTypeParameter(
7          typeof(IReflectiveBaseModelConfiguration<>), baseType))
8      {
9          // load the base model using the explicit interface implementation
10         // we have to do some trickery to get the correct method as it's name is compiler generated.
11         string methodName = string.Format(BaseModelConfigInfoForReflection_DontChange.RuntimeMethodName,
12             ↳ baseType.FullName);
13         // we can't filter by arguments as the generic type is not known yet
14         MethodInfo? baseConfigure = baseType.GetMethod(methodName, BindingFlags.Static |
15             ↳ BindingFlags.NonPublic);
16         if (baseConfigure is not null)
17         {
18             // we have the correct method, make it generic to match the child class
19             MethodInfo genericBaseConfigure = baseConfigure.MakeGenericMethod(entity.Type);
20             // invoke it with the EntityTypeBuilder<T> instance where T is the child class.
21             genericBaseConfigure.Invoke(null, parameters);
22         }
23     }
24     baseType = baseType.BaseType;
25 }

```

Listing 19: Reflectively loading and invoking base class configuration code.

Listing 19 demonstrates the reflective invocation of base class configuration methods. The inheritance chain is traversed upwards until a type directly implementing the base model interface with a generic type parameter that matches the current type is found (lines 6–7). Reflection is then used to retrieve the `ConfigureBaseModel()` method using its compiler-generated name, as the interface is implemented explicitly. Once retrieved, the method is made generic using the concrete entity type as a type parameter and invoked with the matching concrete entity type builder.

C.1.4 Policy enforcement

The last step of the reflective configuration process involves enforcing the naming and mapping policies described in subsection 4.1.4:

```

EntityTypeBuilder entityTypeBuilder = (EntityTypeBuilder)entityTypeBuilderObj;
mappingPolicy.Audit(entityTypeBuilder.Metadata);
namingPolicy.Audit(entityTypeBuilder.Metadata);

```

Listing 20: Enforcing naming and mapping policies.

Since all generic entity type builders share a non-generic base class, the `IMutableEntityType` instance can be retrieved from the entity type builder using polymorphism and passed to both policy objects, as shown in listing 20.

C.2 Iterative accessor optimization

The following sections outline the iterative development process of the parameter accessor generation based on the known `PropertyInfo` metadata.

C.2.1 Late-bound property access

The most straightforward approach to providing late-bound property access is through the `PropertyInfo::GetValue()` and `PropertyInfo::SetValue()` methods. Internally, these methods use reflection and try to retrieve the compiler-generated getter and setter methods using `RuntimePropertyInfo::GetPropertyMethod()` [NET22s, ll. 196, 263, 319] which itself depends on the specific implementation of the `RuntimePropertyInfo` class to use a `MetadataReader` [NET22h, ll. 59, 136] to retrieve the correct method handles [NET22n, l. 1240] [NET22q, ll. 72, 90] [NET20c, l. 1461] of the accessors. This process is time-consuming and results in significant performance overhead, as indicated by the benchmarks shown in appendix D.2.

```
public static class NaivePropertyAccessorFactory
{
    public static ParameterGetter CreateGetter(PropertyInfo pinfo) =>
        (context) => pinfo.GetValue(context);

    public static ParameterSetter CreateSetter(PropertyInfo pinfo) =>
        (context, value) => pinfo.SetValue(context, value);
}
```

Listing 21: Late-bound property access using reflection.

In addition to the significant performance overhead of reflection described earlier, the naive approach shown in listing 21 yields another drawback: it only supports writable and init-only properties (because no setter exists for read-only properties), which violates requirement 11. However, this drawback can be overcome by setting the compiler-generated backing field of the `auto-property` rather than the property itself.

```

public static class NaivePropertyAccessorFactory
{
    public static ParameterGetter CreateGetter(PropertyInfo pinfo) =>
        (context) => pinfo.GetValue(context);

    public static ParameterSetter CreateSetter(PropertyInfo pinfo)
    {
        FieldInfo pBackingField = pinfo.GetBackingField()
            ?? throw new Exception("Could not find backing field of auto-property!");
        (context, value) => pBackingField.SetValue(context, value);
    }
}

static class Extensions
{
    public static FieldInfo? GetBackingField(this PropertyInfo propertyInfo) =>
        propertyInfo.DeclaringType?.GetField($"<{propertyInfo.Name}>k__BackingField",
            BindingFlags.Instance | BindingFlags.NonPublic);
}

```

Listing 22: Late-bound property access using reflection and compiler-generated backing fields.

The approach shown in listing 22 uses reflection to retrieve the compiler-generated backing field by its generated name and instead sets the value of this field. Even though [ECMA-334](#) does not specify an explicit naming convention for compiler-generated backing fields [[ECM22](#), pp. 387–388], [Roslyn](#) follows the naming convention `<PropertyName>k__BackingField` as shown in listing 22. This naming scheme is unlikely to change in the future, as big frameworks like [EF Core](#) rely on it [[NET22a](#), l. 161].

C.2.2 Expressions

Another approach that was abandoned involved the use of [LINQ](#) expressions. The idea was to use [expression trees](#) to emit type-safe, early-bound property accessors by compiling the [expression trees](#) at runtime. This approach achieved near-perfect performance for get-accessors, but set-accessors still required late binding using reflection due to access verification performed by `Expression.Assign()`. The excerpt shown in listing 23 uses an [expression tree](#) to create a getter⁸.

```

public static ParameterGetter CreateGetter(PropertyInfo pinfo)
{
    // create the expression tree for the getter
    // (object owner) => (object)((OwnerType)owner).Property;
    ParameterExpression context = Expression.Parameter(typeof(object), nameof(context));
    Expression<ParameterGetter> getterExpression = Expression.Lambda<ParameterGetter>(
        Expression.Convert(
            Expression.Property(
                Expression.Convert(context, _ownerType),
                _pInfo),
            typeof(object)),
        context);
    // compile the expression tree into IL and return the delegate
    return getterExpression.Compile();
}

```

Listing 23: Excerpt of early bound property access using [expression trees](#).

⁸ The full implementation is provided in listing 28 on page 70.

C.2.3 Reflection emit

The final approach uses the [Reflection.Emit API](#) to generate [IL](#) byte code that directly reads and writes to the property, eliminating the dependency on reflection and bypassing access validation. This solution is able to match the performance of direct property access while still providing the flexibility of determining (generating) the concrete implementation of property accessors at runtime. After emission, this implementation is almost indistinguishable from a accessors compiled by [Roslyn](#) in terms of performance and runtime behavior, as indicated by the benchmarks in appendix [D.2](#).

```
public static ParameterSetter CreateSetter(PropertyInfo pInfo)
{
    FieldInfo backingField = pInfo.GetBackingField()
    ?? _throwHelper.Throw<InvalidOperationException, FieldInfo>($"...");
    DynamicMethod dynamicMethod = new($"Setter_{Guid.NewGuid:N}",
        typeof(void), _parameterSetterArgumentTypes,
        _ownerType, true);
    ILGenerator generator = dynamicMethod.GetILGenerator();
    generator.Emit(OpCodes.Ldarg_0);
    generator.Emit(OpCodes.Castclass, _ownerType);
    generator.Emit(OpCodes.Ldarg_1);
    generator.Emit(OpCodes.Unbox_Any, _pInfo.PropertyType);
    generator.Emit(OpCodes.Stfld, backingField);
    generator.Emit(OpCodes.Ret);
    return dynamicMethod.CreateDelegate<ParameterSetter>();
}
```

Listing 24: Excerpt of raw [IL](#) byte code generation using the [Reflection.Emit API](#).

The excerpt⁹ in listing 24 illustrates the creation and use of a dynamic method to emit [IL](#) byte code that sets the value of a property. First, a `DynamicMethod` instance of type `void` with two `object` parameters is created, with the first parameter being the target instance of the [I/O container](#) and the second parameter representing the value to be assigned. The setter name is generated randomly at runtime to prevent naming conflicts. An `ILGenerator` instance is then obtained from the dynamic method to emit [IL](#) instructions. The emitted [IL](#) code loads both arguments onto the stack, casting the first argument (the [I/O container](#)) and unboxing the second argument to their respective concrete types. The unboxed value is then assigned to the backing field in the [I/O container](#) object, and a `delegate` is created from the dynamic method.

⁹ The full implementation can be found in listing 29 on page 73.

D Performance benchmarks

This appendix chapter presents the performance benchmarks conducted for this work.

D.1 Methodology

All benchmarks were run on a Dell Vostro 5590 laptop with an Intel Core i7-10510U@1.80GHz CPU and 16GB RAM running Windows 10 Pro (Version 21H2, OS Build 19044.2364). The computer was plugged into a power outlet during testing, and a well-ventilated environment was provided to ensure that no thermal throttling occurred due to heat buildup. The benchmarks were performed using [BenchmarkDotNet](#).

D.2 Property accessor optimization

A benchmark session was conducted to evaluate the optimal approach for generating property accessors. The benchmark compares the performance of the naive “*Reflected*” approach using reflection, as shown in listing 22 on page 59, against [expression tree](#)-compilation (“*Expression Tree*”), as shown in listing 23 on page 59, as well as raw IL byte code emission (“*IL Emitted*”) using the [Reflection.Emit API](#), as shown in listing 29 on page 73. “*Direct*” early-bound property access with perfect knowledge about the concrete type of the [I/O container](#) object is used as a baseline, which is shown in listing 25. All benchmarks were run for accessing reference and value types.

```
private static object DirectGetterImplValueType(object dummy) =>
    ((Dummy)dummy).ValueType;
private static void DirectSetterImplValueType(object dummy, object i) =>
    ((Dummy)dummy).ValueType = (int)i;
```

Listing 25: “*Direct*” early-bound property access.

Benchmarks were run as a `LongRunJob` on both getters and setters with a `WarmupCount` batch size of 15 and an `IterationCount` of 100 batches. The benchmark was repeated three times.

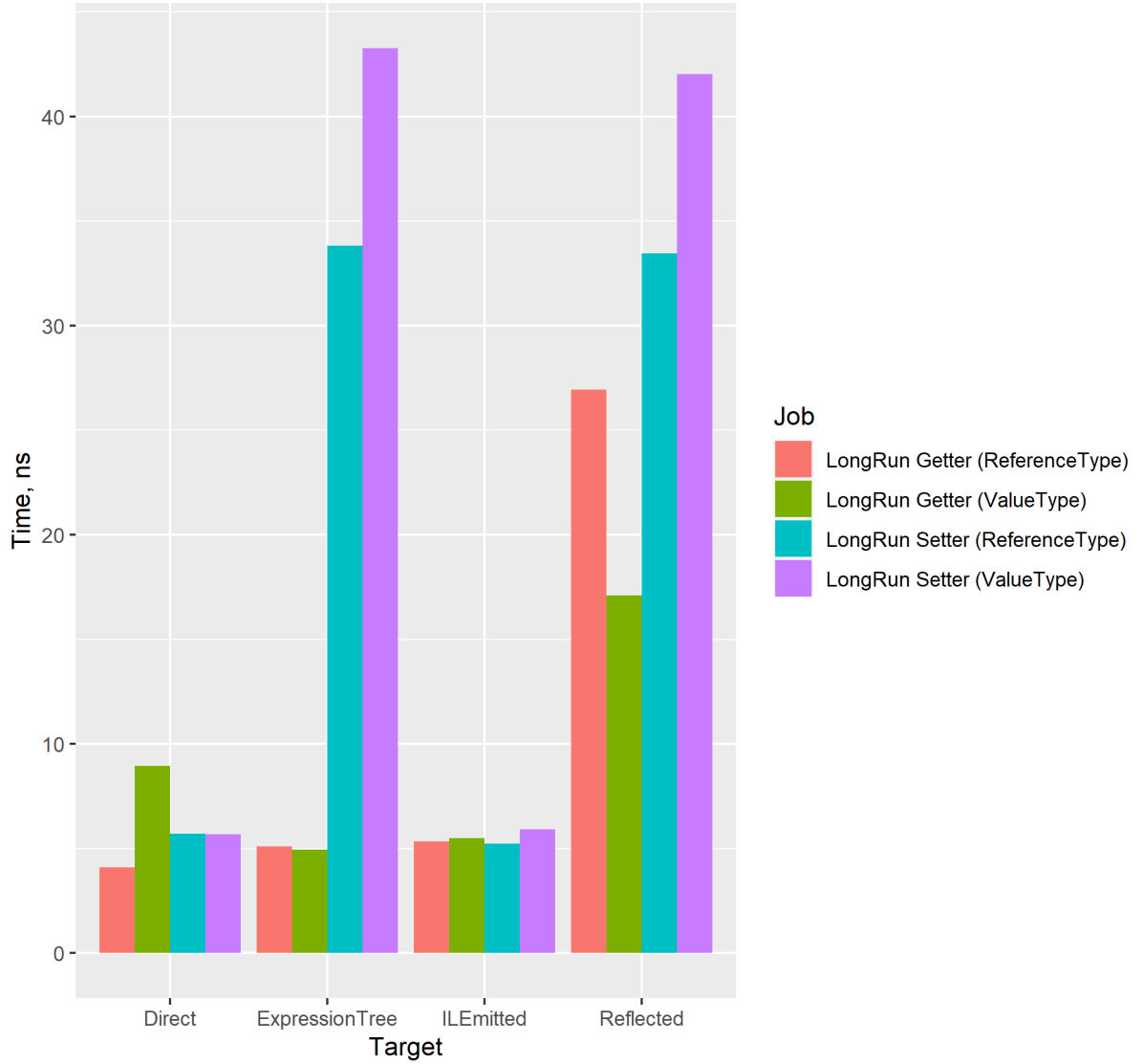


Figure D.1: Benchmark results for property accessor generation.

The results of the benchmarks are visualized as a bar plot in figure D.1. For every target approach (“Direct”, “Expression Tree”, “IL Emitted”, and “Reflected”), the mean execution time of the getter and setter for both reference and value types is shown. The execution time of direct early-bound access is used as a baseline, as it was expected to be the fastest approach.

Consistent with the predictions, reflection-based accessors are the slowest overall, being, on average, 5.4 times slower than direct early-bound access. However, benchmarks performed with fewer iterations indicate that the slowdown is even more significant for smaller iteration counts. The expression tree-based approach is as fast as direct access for getters while being even slower than the naive reflection-based approach for setters. This can be explained by the aforementioned access validation performed by `Expression.Assign()` and the required reflection-based bypass. Due to manual IL instruction-level micro-optimizations, the

approach using raw [IL](#) byte code emission via the [Reflection.Emit API](#) is, on average, even 3% faster than direct early-bound access, which is a remarkable result, as the approach is still (technically) late bound, as no knowledge about the concrete type of the [I/O container](#) object si available during compile time.

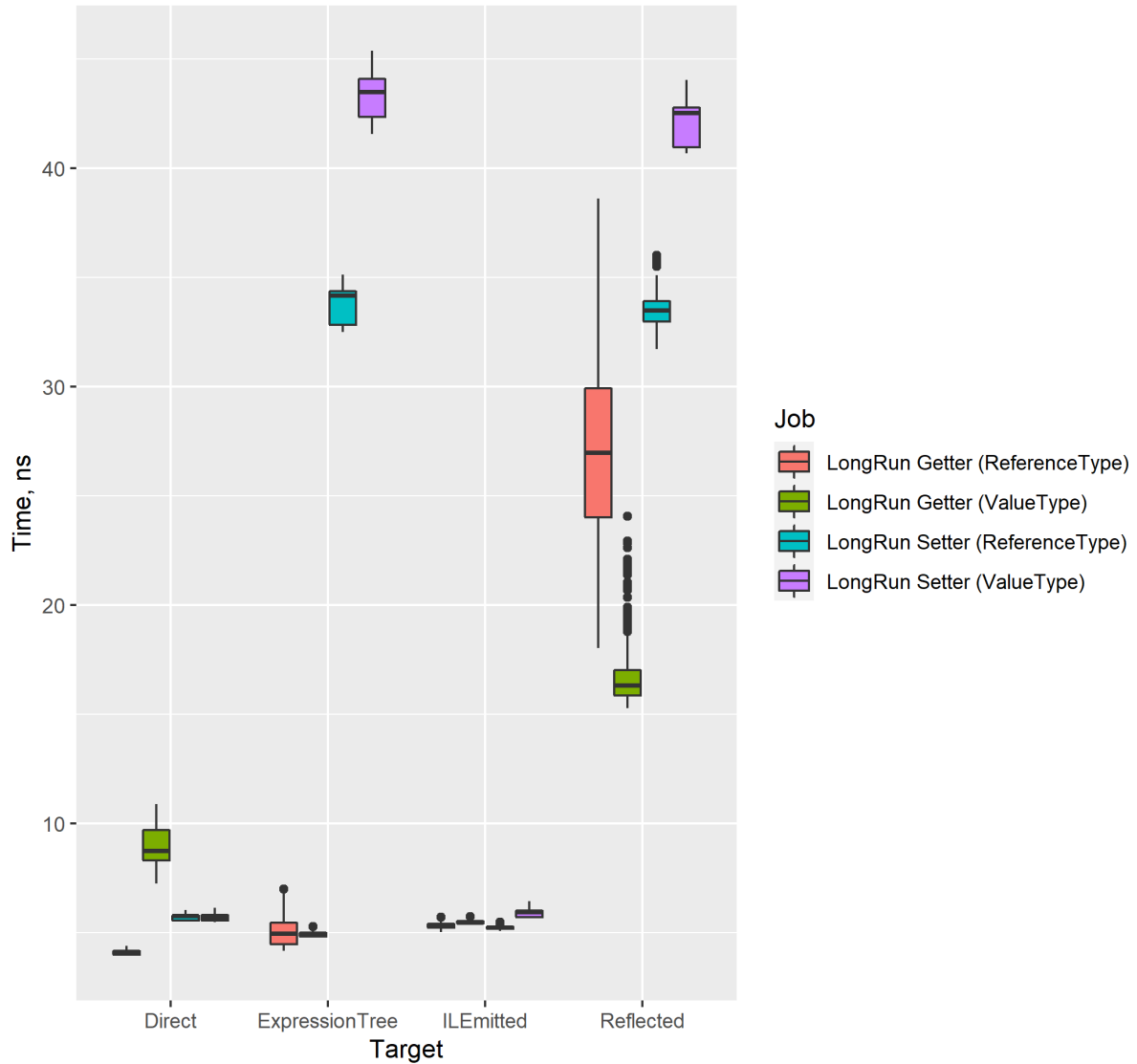


Figure D.2: Box plot for property accessor generation.

The box plot shown in figure [D.2](#) shows the execution time of all approaches and their variation. The plot indicates that apart from being the fastest, the “*IL Emitted*” approach is also the most consistent, showing only minimal variation in execution time.

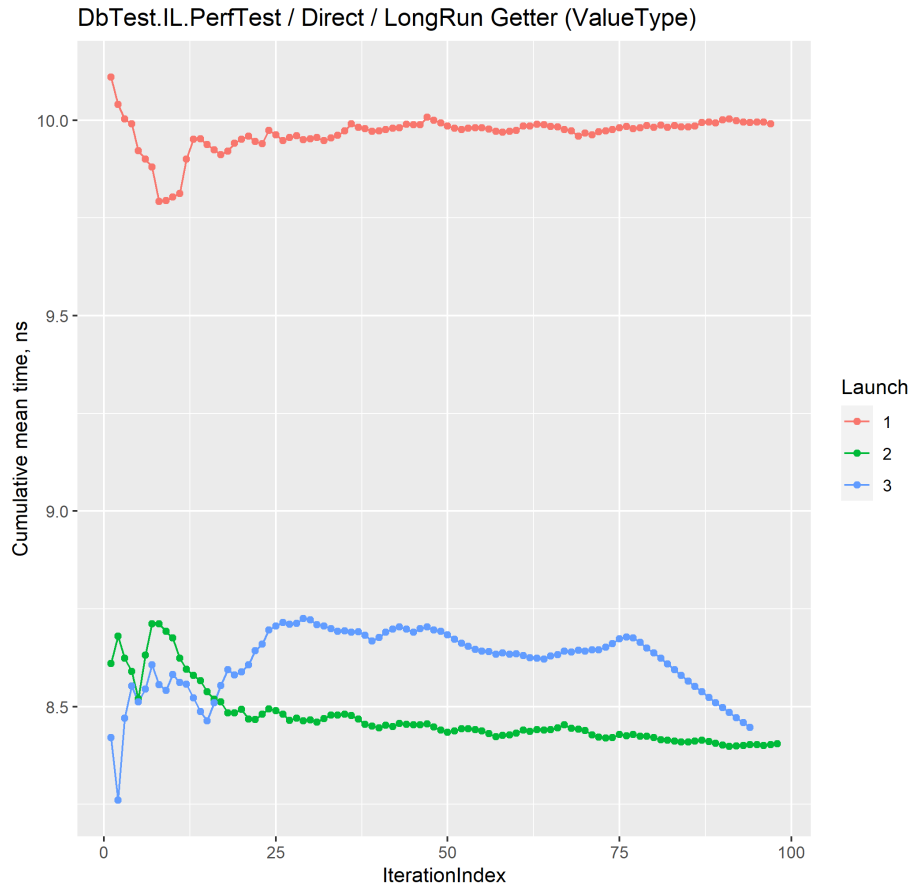


Figure D.3: Measurement inaccuracies in mean execution time of the direct getter for value types.

The significantly higher deviation in execution times of the direct getter for value types can be explained by measurement inaccuracies due to background processes, as execution times differ considerably between the first and subsequent benchmark runs, as indicated by figure D.3.

The largest range of values can be observed for the reflection-based setter of reference types. This may be due to polymorphic behavior and virtual method calls of the underlying reflection implementation; however, this anomaly was not investigated further.

All in all, the “*IL Emitted*” approach was selected as the optimal solution to generate property accessors.

D.3 Overall performance evaluation

```
[RPlotExporter]
[LongRunJob]
public class ProcMappingBenchmark
{
    private readonly MySqlDbContext _mySqlDbContext;
    private readonly MySqlTestProcedure _cachedProcedure;
    public ProcMappingBenchmark()
    {
        const string mySqlConnectionString = "Server=localhost;Database=test;...";
        DbContextOptionsBuilder<MySqlDbContext> mySqlBuilder = new();
        mySqlBuilder.UseMySQL(mySqlConnectionString);
        _mySqlDbContext = new MySqlDbContext(mySqlBuilder.Options);
        _mySqlDbContext.ChangeTracker.DetectChanges(); // invoke context
        ↪ initialization before first benchmark
        _cachedProcedure = _mySqlDbContext.Procedure<MySqlTestProcedure>();
    }
    [Benchmark]
    public int Recap()
    {
        int a = 1;
        return _mySqlDbContext.Procedure<MySqlTestProcedure>().Invoke(ref a, out int
        ↪ b) + a + b;
    }
    [Benchmark]
    public int RecapCached()
    {
        int a = 1;
        return _cachedProcedure.Invoke(ref a, out int b) + a + b;
    }
    [Benchmark(Baseline = true)]
    public int Raw()
    {
        MySqlParameter a = new("a", MySqlDbType.Int32)
        {
            Value = 1,
            Direction = ParameterDirection.InputOutput
        };
        MySqlParameter b = new("b", MySqlDbType.Int32)
        {
            Direction = ParameterDirection.Output
        };
        DbConnection connection = _mySqlDbContext.Database.GetDbConnection();
        if (connection.State is ConnectionState.Closed)
        {
```

```

        connection.Open();
    }
    using DbCommand cmd = connection.CreateCommand();
    cmd.CommandText = "test";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(a);
    cmd.Parameters.Add(b);
    return ((int)cmd.ExecuteScalar(!) + (int)a.Value + (int)b.Value;
}
}

```

Listing 26: Performance benchmark for stored procedure invocation using [RECAP](#) and raw SQL.

Listing 26 shows the benchmark used to evaluate the performance of stored procedure invocation using [RECAP](#) and raw [ADO.NET](#) database commands. The “*Recap*” method uses the [RECAP](#) framework to invoke the procedure and then returns the sum of all output parameters. This is done to safeguard against the compiler’s dead-code elimination affecting the benchmark methods. The “*RecapCached*” method caches the [execution context](#) instance and uses it to invoke the procedure. The “*Raw*” method directly invokes the stored procedure using [ADO.NET](#) and returns the sum of all output parameters. The benchmark was run on a MySQL database hosted locally in a Docker container with the test procedure provided in listing 27.

```

CREATE PROCEDURE test(INOUT a INT, OUT b INT)
BEGIN
    DECLARE c INT;
    SET b = a;
    SET a = 314159265; -- first 9 digits of pi, no meaning behind this.
    SET c = -271828182; -- first 9 digits of e * (-1)
    SELECT c;
END;

```

Listing 27: Stored procedure used for benchmarking.

The stored procedure sets the output parameters to the input parameters and returns a constant value. The benchmarks were run as a `LongRunJob`, launching the benchmark three times, executing each method in 100 bunches of 4096 iterations, and an additional 15 warmup bunches before each run. Therefore each benchmark method was executed 1,413,120 times in total. The baseline of the measurement was set to the raw invocation. The results of this benchmark are shown in figure [D.4](#) on the next page.

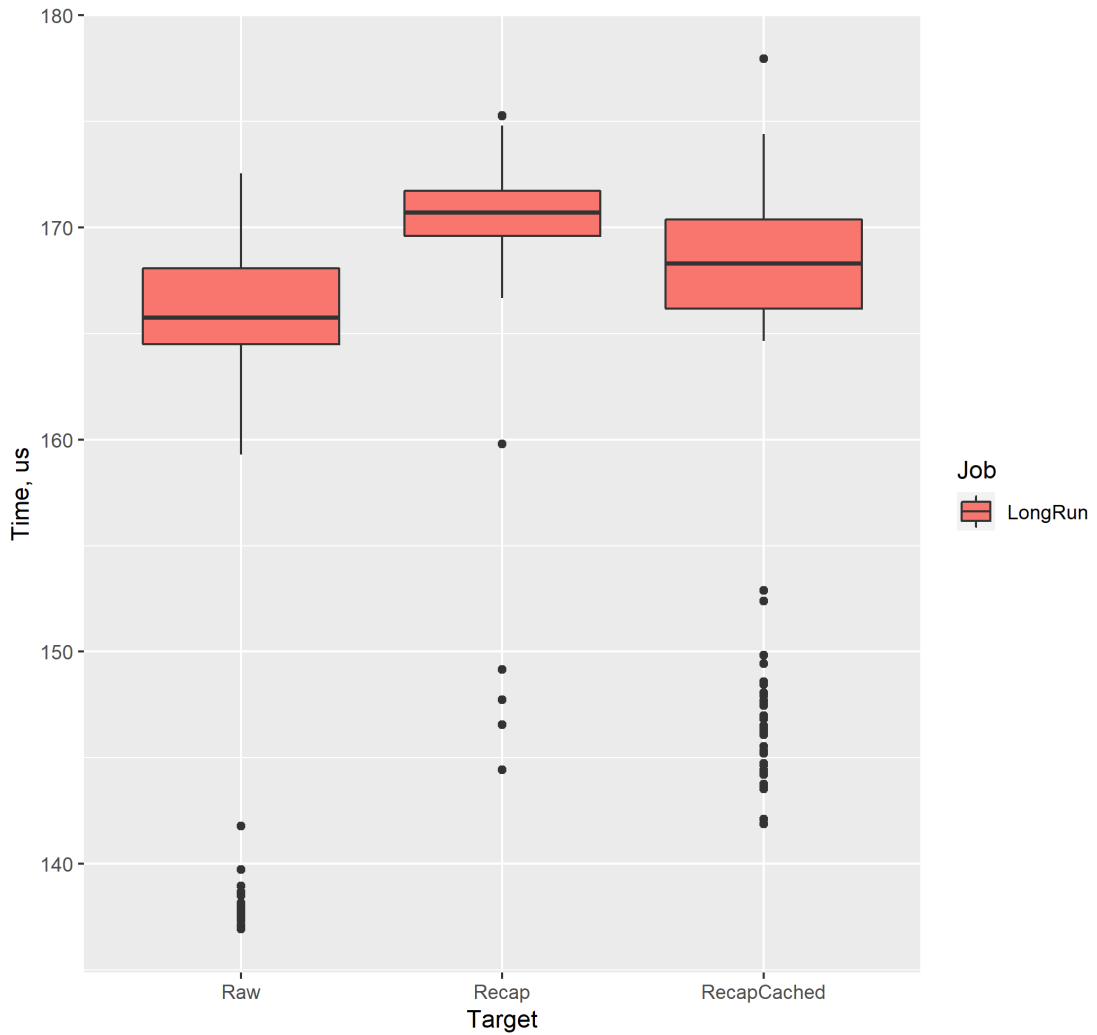


Figure D.4: Box plot displaying the measured performance overhead of [RECAP](#) compared to direct invocation.

As depicted in figure [D.4](#), the results of the benchmark reveal that the “*Raw*” box plot exhibits the highest speed, with a range of values extending from 158 μ s to 172.5 μ s. Conversely, the “*Recap*” box plot demonstrates a slower overall performance but displays a considerably reduced spread, with values ranging between 167 μ s and 175 μ s. The “*RecapCached*” box plot exhibits a faster overall performance than the “*Recap*” box plot but indicates a more extensive interquartile range and a slight left skewness in its distribution. The overall range of values for this data set is 164.5 μ s to 174.4 μ s. The raw invocation is slightly faster than the [RECAP](#) framework, but the differences are all within the margin of error.

Overall, the results of this benchmark demonstrate that the [RECAP](#) framework incurs minimal performance overhead when invoking stored procedures. The results also indicate that caching the [execution context](#) in-between subsequent invocations can improve performance as existing `DbParameter` objects may be re-used, depending on the database engine.

Table D.1 provides a textual representation of these benchmark results.

Table D.1: Benchmark results (execution times and memory allocations).

| Method | Mean | Error | StdDev | Median | Ratio | Allocated | Alloc Ratio |
|-------------|---------------|--------------|--------------|---------------|-------|-----------|-------------|
| Recap | 170.4 μ s | 0.63 μ s | 3.25 μ s | 170.7 μ s | 1.04 | 12.16 KB | 1.01 |
| RecapCached | 166.2 μ s | 1.46 μ s | 7.52 μ s | 168.3 μ s | 1.01 | 11.68 KB | 0.97 |
| Raw | 164.3 μ s | 1.55 μ s | 8.05 μ s | 165.8 μ s | 1.00 | 12.00 KB | 1.00 |

Allocation statistics in table D.1 show that the RECAP framework allocates marginally more memory than the direct ADO.NET invocation (12.16 KB > 12.00 KB), with cached execution context instances allocating slightly less memory than non-cached ones (11.68 KB < 12.16 KB). However, the differences in memory allocation are minimal and can be considered negligible as most of the allocated memory is used by the underlying database provider. Overall, the overhead in execution time of using RECAP is generally insignificant when interacting with remote database engines, as network latency alone is often several orders of magnitude larger (1ms to 100ms \gg 6 μ s) than the overhead induced by RECAP's PCO, I/O container, and execution context management.

E Full implementations

This appendix chapter provides the complete implementations of some of the features presented in earlier sections of this work.

E.1 Expression tree-based accessors

```
internal readonly struct ExpressionTreeAccessorBuilder : IAccessorBuilder, IAccessorBuilderFactory
{
    private static readonly Type[] _fieldSetValueParameterTypes = new Type[] { typeof(object),
    ↪  typeof(object) };

    private readonly PropertyInfo _pInfo;
    private readonly Type _ownerType;
    private readonly IThrowHelper _throwHelper;

    private ExpressionTreeAccessorBuilder(PropertyInfo propertyInfo, IThrowHelper throwHelper)
    {
        _pInfo = propertyInfo;
        _throwHelper = throwHelper;
        _ownerType = _pInfo.DeclaringType!;
    }

    public ParameterGetter BuildGetter()
    {
        // create a parameter expression for the context
        ParameterExpression context = Expression.Parameter(typeof(object), nameof(context));
        // create the expression tree for the getter
        // this is equivalent to:
        // (object owner) => (object)(((OwnerType)owner).Property);
        Expression<ParameterGetter> getterExpression = Expression.Lambda<ParameterGetter>(
            // convert the property value to object
            Expression.Convert(
                // get the property value
                Expression.Property(
                    // convert the owner to the owner type
                    Expression.Convert(context, _ownerType),
                    // get the property info of Property
                    _pInfo,
                    typeof(object)),
                context);
        // compile the expression tree into IL and return the delegate
        return getterExpression.Compile();
    }

    public ParameterSetter BuildSetter()
    {
        // create a parameter expression for the context
        ParameterExpression contextParam = Expression.Parameter(typeof(object), nameof(contextParam));
        // get the backing field for the property
        // this is the field that is automatically generated by the compiler when you use the auto property
        ↪  syntax (int Foo { get; set; })
        // if the property does not have an auto-generated backing field we throw an exception (we do not
        ↪  support manually implemented properties)
        FieldInfo backingField = _pInfo.GetBackingField()
```

```

        // throw an InvalidOperationException if there is no backing field
        ?? _throwHelper.Throw<InvalidOperationException>,
        ↪ FieldInfo>($"Parameter is an output parameter, but linked property {_pInfo.Name}"
        + " does not have a valid auto-generated backing field.");
    // get the setter for the backing field
    MethodInfo backingFieldSetter = typeof(FieldInfo)
        .GetMethod(
            nameof(FieldInfo.SetValue),
            BindingFlags.Instance | BindingFlags.Public,
            _fieldSetValueParameterTypes!);
    // create a parameter expression for the value
    ParameterExpression valueParam = Expression.Parameter(typeof(object), nameof(valueParam));
    // create the expression tree for the setter
    // this is equivalent to:
    // (object owner, object value) => backingField.SetValue(owner, value);
    // we cannot directly assign the value to the backing field as we did in IL, because .NETs
    ↪ reflection code is smart and determines the field
    // to be readonly (for get or init properties) and then just dies :P
    // so we have to do some inception-level reflection craziness and reflectively call the reflected
    ↪ backing field's setter, because that for some reason works...
    Expression<ParameterSetter> setterExp = Expression.Lambda<ParameterSetter>(
        Expression.Call(
            // get the backing field as a constant
            Expression.Constant(backingField),
            // get the backing field setter
            backingFieldSetter,
            // get the context parameter
            contextParam,
            // get the value parameter
            valueParam),
        // set the parameter names
        contextParam,
        valueParam);
    // compile the expression tree into a delegate
    return setterExp.Compile();
}

public static IAccessorBuilder CreateBuilder(PropertyInfo propertyInfo, IThrowHelper throwHelper) =>
    ↪ new ExpressionTreeAccessorBuilder(propertyInfo, throwHelper);
}

```

Listing 28: Early bound property access using expressions.

E.2 IL emitted accessors

```

internal readonly struct ILAccessorBuilder : IAccessorBuilder, IAccessorBuilderFactory
{
    private static readonly Type[] _parameterSetterArgumentTypes = new Type[] { typeof(object),
    ↪ typeof(object) };
    private static readonly Type[] _parameterGetterArgumentTypes = new Type[] { typeof(object) };
    private static readonly MethodInfo _unsafeAs = typeof(Unsafe)
        .GetMethod(nameof(Unsafe.As), 1, TypeArray.Of<object>());
    private static readonly MethodInfo _unsafeAsFromTo = typeof(Unsafe)
        .GetMethods(BindingFlags.Static | BindingFlags.Public)
        .Where(m => m.Name is nameof(Unsafe.As) && m.GetGenericArguments().Length is 2)
        .Single();
}

```

```

private readonly PropertyInfo _pInfo;
private readonly Type _ownerType;
private readonly IThrowHelper _throwHelper;

private ILAccessorBuilder(PropertyInfo propertyInfo, IThrowHelper throwHelper)
{
    _pInfo = propertyInfo;
    _ownerType = propertyInfo.DeclaringType!;
    _throwHelper = throwHelper;
}

public ParameterGetter BuildGetter()
{
    // create a new dynamic method for the getter
    // the method name uses a random GUID, to prevent conflicts.
    // we declare the method on the owner type; this is not strictly necessary, but it keeps things
    // clean. in C# this would be equivalent to:
    // private static object Getter_acb9b0c03b9d4b9e9b0c0b3b9d4b9e9b(object owner) =>
    //     (object)((OwnerType)owner).Property);
    DynamicMethod dynamicMethod = new(
        $"Getter_{Guid.NewGuid:N}",
        typeof(object),
        _parameterGetterArgumentTypes,
        _ownerType,
        true);
    // we use Unsafe.As<OwnerType>(owner) instead of (OwnerType)owner because Castclass is sloooooow
    MethodInfo unsafeAs1 = _unsafeAs.MakeGenericMethod(_ownerType);
    // get the IL generator for the dynamic method
    ILGenerator generator = dynamicMethod.GetILGenerator();
    // load the first argument (the owner) onto the stack
    generator.Emit(OpCodes.Ldarg_0);
    // cast the owner to the owner type
    generator.Emit(OpCodes.Call, unsafeAs1);
    // call the getter method of the property, which will leave the value of the property on the stack
    generator.Emit(OpCodes.Callvirt, _pInfo.GetMethod!);
    // if the property type is a value type, we need to box it to an object
    // otherwise, we can just return the value
    if (_pInfo.PropertyType.IsValueType)
    {
        generator.Emit(OpCodes.Box, _pInfo.PropertyType);
    }
    // return the value on the stack
    generator.Emit(OpCodes.Ret);
    // create a delegate for the dynamic method
    return dynamicMethod.CreateDelegate<ParameterGetter>();
}

public ParameterSetter BuildSetter()
{
    // get the backing field for the property
    // this is the field that is automatically generated by the compiler when you use the auto property
    // syntax (int Foo { get; set; }). if the property does not have an auto-generated backing field we
    // throw an exception (we do not support manually implemented properties)
    FieldInfo backingField = _pInfo.GetBackingField()
        // if we can't find the backing field, throw an exception
        ?? _throwHelper.Throw<InvalidOperationException, FieldInfo>($"Parameter is an output parameter"
            + " but linked property {_pInfo.Name} does not have a valid auto-generated backing field");
    // in C# this would be equivalent to:

```

```

// private static void Setter_acb9b0c03b9d4b9e9b0c0b3b9d4b9e9b(object owner, object value) =>
//     ((OwnerType)owner).Property = (PropertyType)value;
// except that we use the backing field instead of the property which allows us to set the value
// even if the property is read-only or init-only (default for records)
DynamicMethod dynamicMethod = new(
    $"Setter_{Guid.NewGuid:N}",
    typeof(void),
    TypeArray.Of<object, object>(),
    _ownerType,
    true);
// we use Unsafe.As<OwnerType>(owner) instead of (OwnerType)owner because Castclass is sloooooow
MethodInfo unsafeAs1 = _unsafeAs.MakeGenericMethod(_ownerType);
// get the IL generator for the dynamic method
ILGenerator generator = dynamicMethod.GetILGenerator();
// load the first argument (the owner) onto the stack
generator.Emit(OpCodes.Ldarg_0);
// cast the owner to the owner type
generator.Emit(OpCodes.Call, unsafeAs1);
// we can optimize manually here.
// this is why in the benchmarks we are faster than Roslyn :)
// Roslyn just always emits unbox.any
if (_pInfo.PropertyType.IsValueType)
{
    // push the value onto the stack (second argument, index 1)
    generator.Emit(OpCodes.Ldarg_1);
    // unboxes object to managed byref pointer
    generator.Emit(OpCodes.Unbox, _pInfo.PropertyType);
    // switch on the type of the property to determine the correct Ldind instruction
    // the Ldind family of instructions loads a value from a managed pointer
    Type t = _pInfo.PropertyType;
    OpCode? simpleIndirectLoad = t switch
    {
        _ when t == typeof(nint) || t == typeof(nuint) => OpCodes.Ldind_I,
        _ when t == typeof(sbyte) => OpCodes.Ldind_I1,
        _ when t == typeof(short) => OpCodes.Ldind_I2,
        _ when t == typeof(int) => OpCodes.Ldind_I4,
        _ when t == typeof(long) || t == typeof(ulong) => OpCodes.Ldind_I8,
        _ when t == typeof(byte) || t == typeof(bool) => OpCodes.Ldind_U1,
        _ when t == typeof(ushort) => OpCodes.Ldind_U2,
        _ when t == typeof(uint) => OpCodes.Ldind_U4,
        _ when t == typeof(float) => OpCodes.Ldind_R4,
        _ when t == typeof(double) => OpCodes.Ldind_R8,
        _ => null
    };
    // if the type is not one of the above, we need to use Ldobj (for larger/not primitive types)
    if (simpleIndirectLoad is null)
    {
        // Ldobj also needs some type information, so we need to push the type token onto the stack
        generator.Emit(OpCodes.Ldobj, t);
    }
    else
    {
        // this is one of the simple Ldind instructions
        // they are already typed, so we don't need to push the type token
        generator.Emit(simpleIndirectLoad.Value);
    }
}
else
{

```

```

// for reference types, we can cast the reference (no Castclass with checks for custom
// conversions needed). in theory we could also just jam the new pointer in there without
// casting, it wouldn't actually matter. but the JIT is smart enough to optimize this
// away anyway. benchmarking shows that this is even faster than just jamming the pointer
// in there probably because this legal IL is easier to jit than the illegal IL we'd get
// otherwise. Therefore we use Unsafe.As<object, TargetType>(ref value) here :)
MethodInfo unsafeAs2 = _unsafeAsFromTo
    .MakeGenericMethod(new Type[] { typeof(object), _pInfo.PropertyType });
// push argument address (ByRef) of the value onto the stack (second argument, index 1, ByRef)
generator.Emit(OpCodes.Ldarga_S, 1);
// invoke Unsafe.As<object, TargetType>(ref value)
generator.Emit(OpCodes.Call, unsafeAs2);
// resolve the ByRef pointer to a normal reference (ByRef<object> is just void**)
generator.Emit(OpCodes.Ldind_Ref);
}

// whatever we have on the stack now is the value we want to set
// we need to store it in the backing field
generator.Emit(OpCodes.Stfld, backingField);
// return from the method
generator.Emit(OpCodes.Ret);
// create a delegate for the dynamic method
return dynamicMethod.CreateDelegate<ParameterSetter>();
}

public static IAccessorBuilder CreateBuilder(PropertyInfo propertyInfo, IThrowHelper throwHelper) =>
    new ILAccessorBuilder(propertyInfo, throwHelper);
}

```

Listing 29: Micro-optimized [IL](#) byte code-based accessor generation using the [Reflection.Emit API](#).