.NET Native AOT

Data Access without Entity Framework Core
but with Humble Objects

Kenny Pflug
@feO2x
Consultant

think
tecture

On the card today

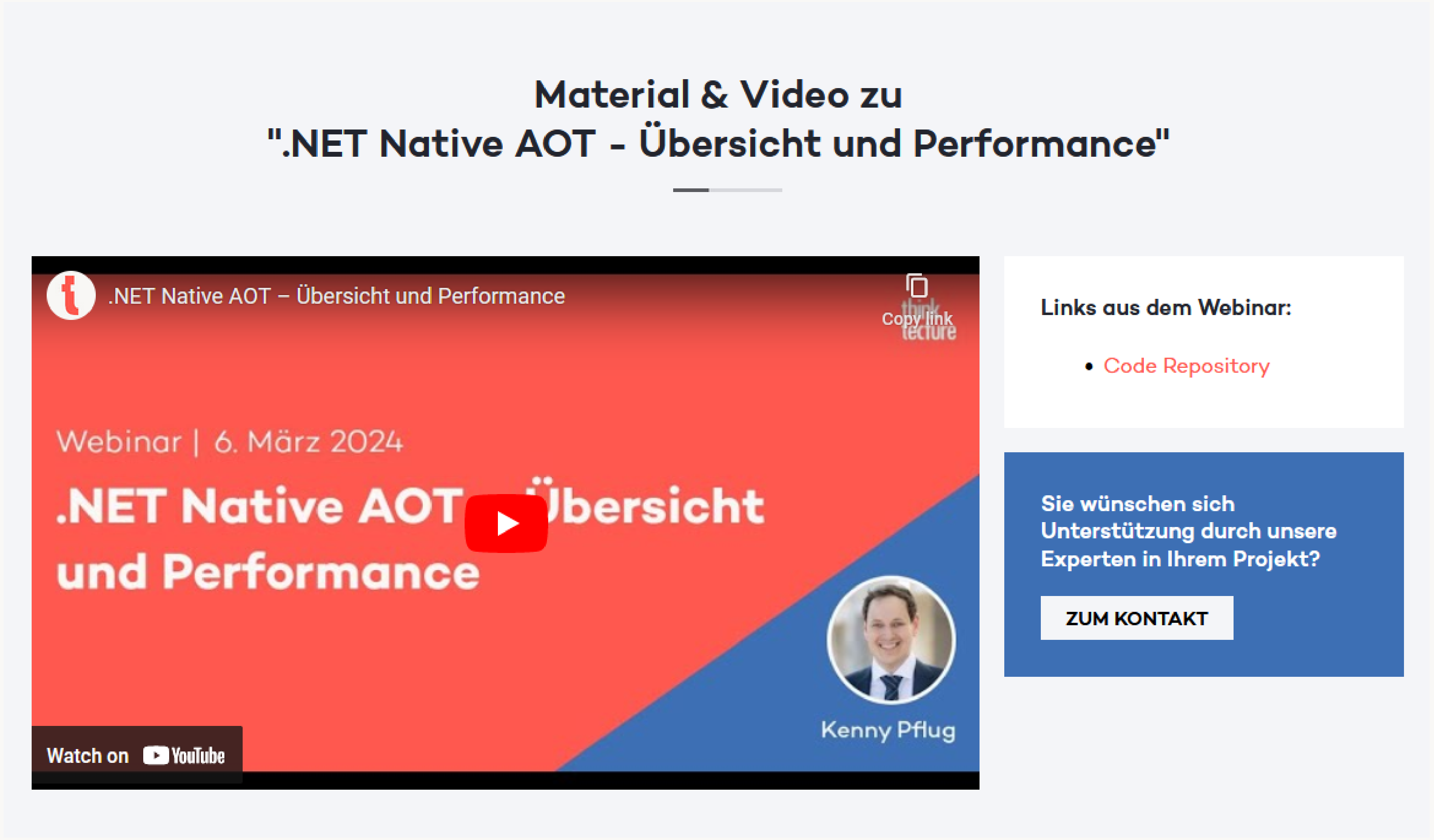
- From DbContext and DbSet to DbConnection, DbCommand, and DbTransaction
- How to isolate them with Humble Objects from your Core Logic?
- How do these relate to the Unit-of-Work and Repository patterns?
- Which ORMs can be used with Native AOT?
- Q&A

Kenny Pflug

Consultant @ Thinktecture AG

- Distributed Systems with ASP.NET Core
- .NET internals
- Cloud-native

kenny.pflug@thinktecture.com @feO2x https://www.thinktecture.com



Material & Video zu ".NET Native AOT - Übersicht und Performance"

.NET Native AOT – Übersicht und Performance
Copy link
Webinar | 6. März 2024
.NET Native AOT – Übersicht und Performance
Kenny Pflug
Watch on YouTube

Links aus dem Webinar:

- Code Repository

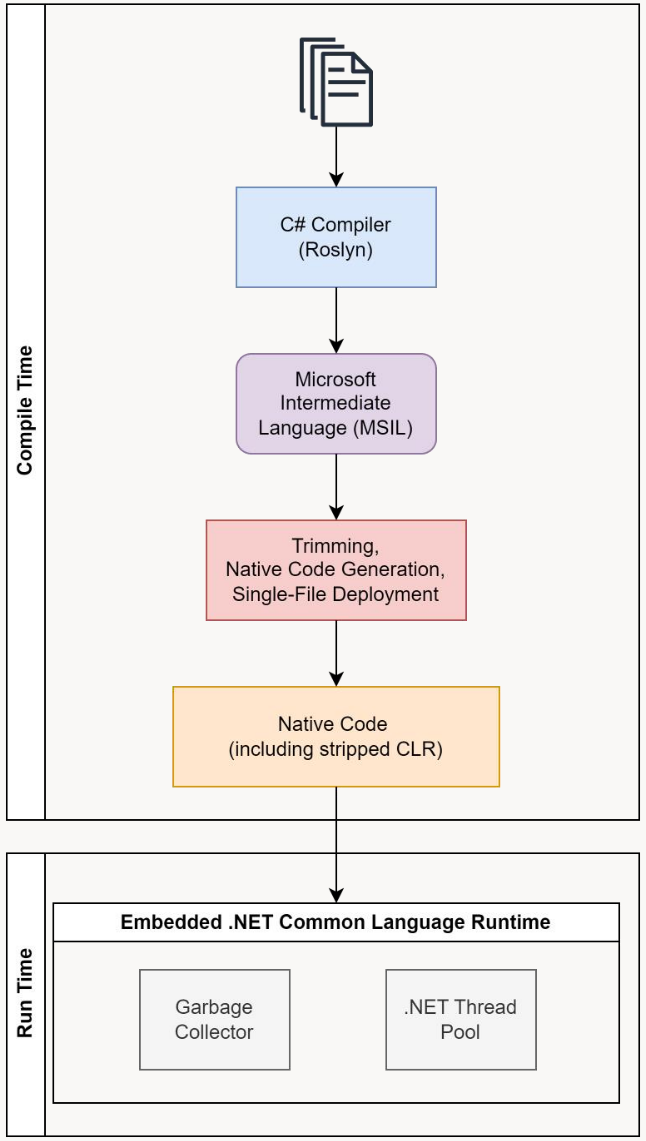
Sie wünschen sich Unterstützung durch unsere Experten in Ihrem Projekt?

ZUM KONTAKT

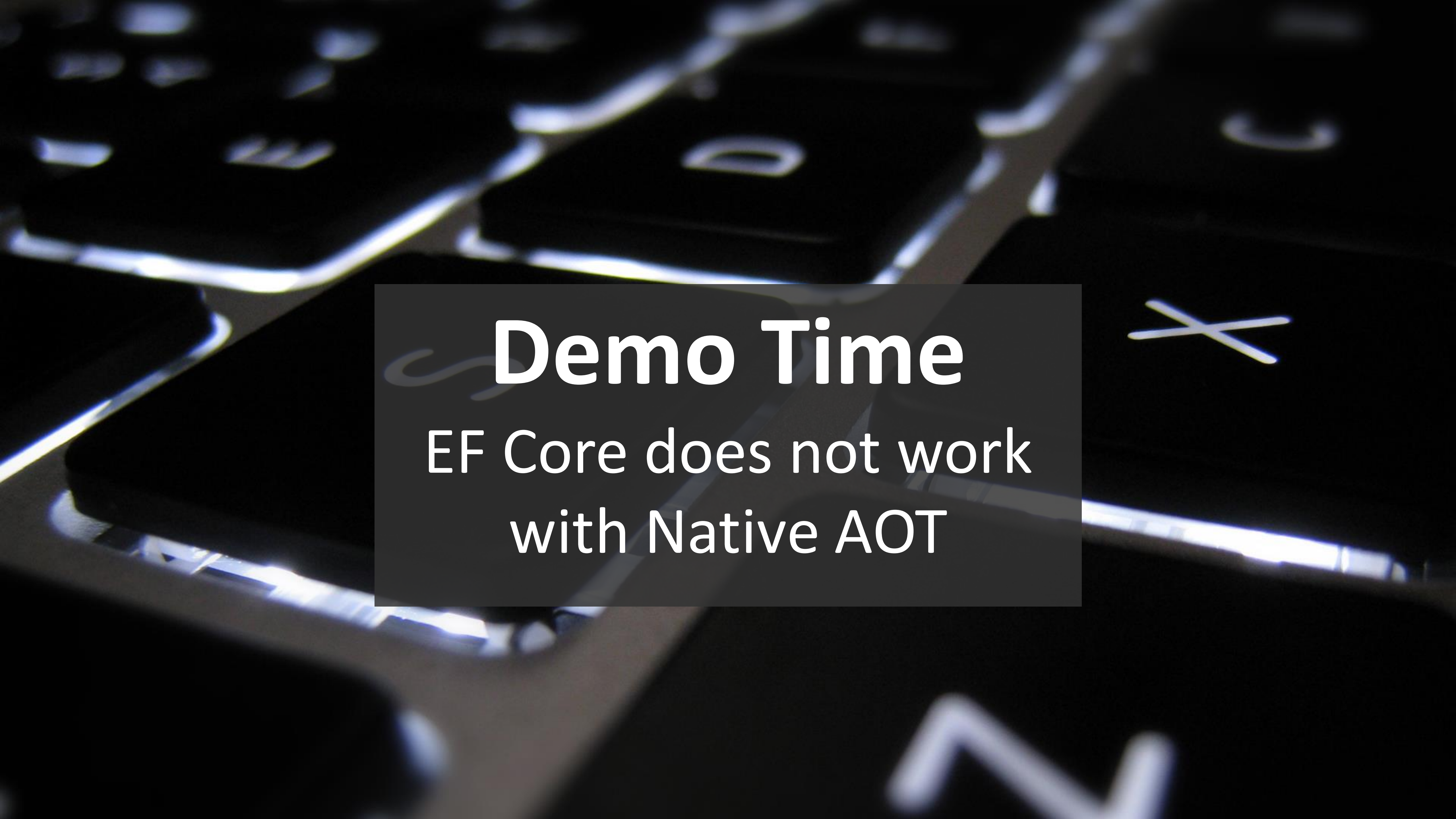
A quick reminder

During Development
Regular CLR

After Publish
Native Code


Compile Time
C# Compiler (Roslyn)
Microsoft Intermediate Language (MSIL)
Trimming, Native Code Generation, Single-File Deployment
Native Code (including stripped CLR)
Run Time
Embedded .NET Common Language Runtime
Garbage Collector
.NET Thread Pool

From EF Core to ADO.NET

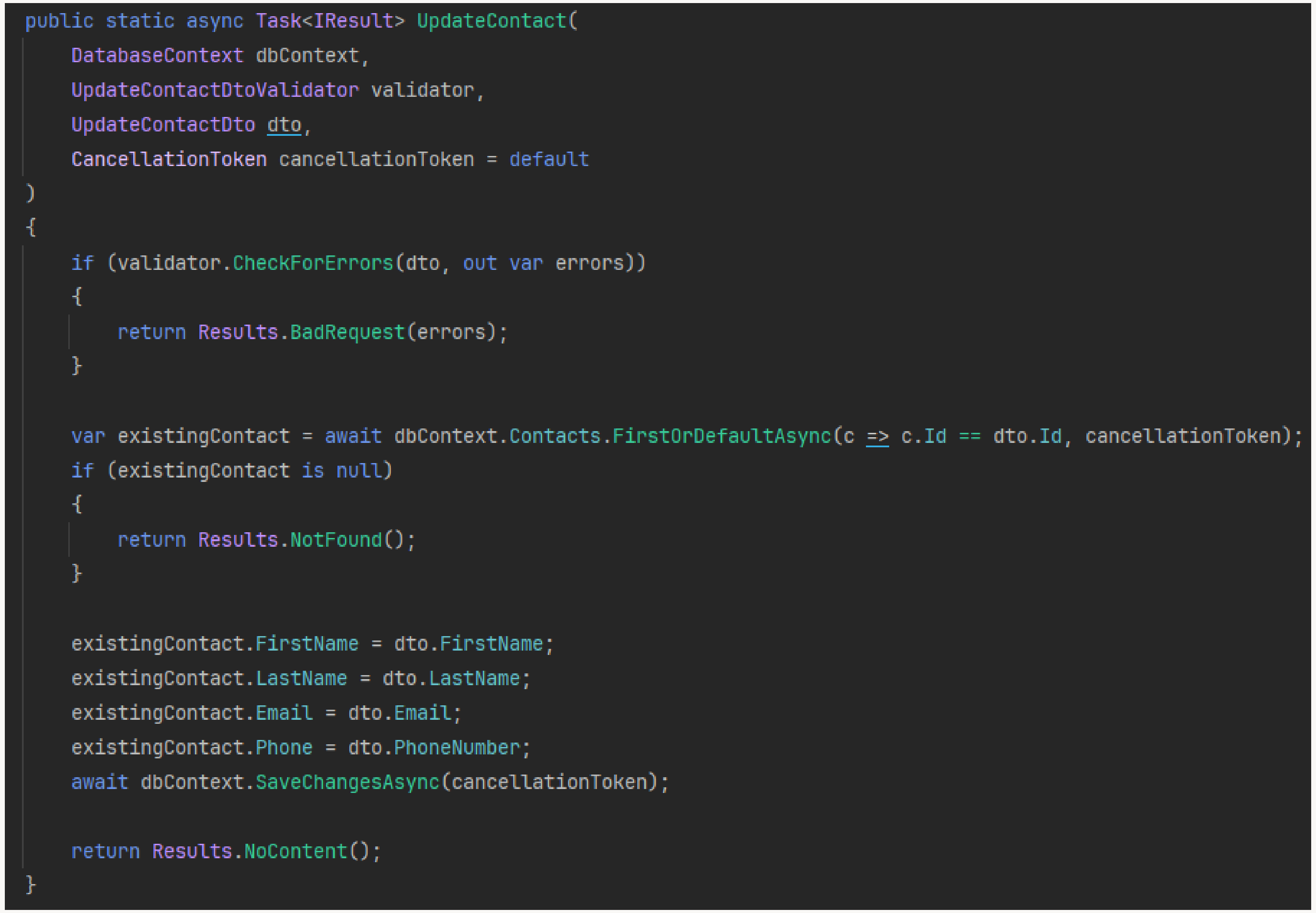


Demo Time

EF Core does not work with Native AOT

What does EF Core do for us behind the covers?

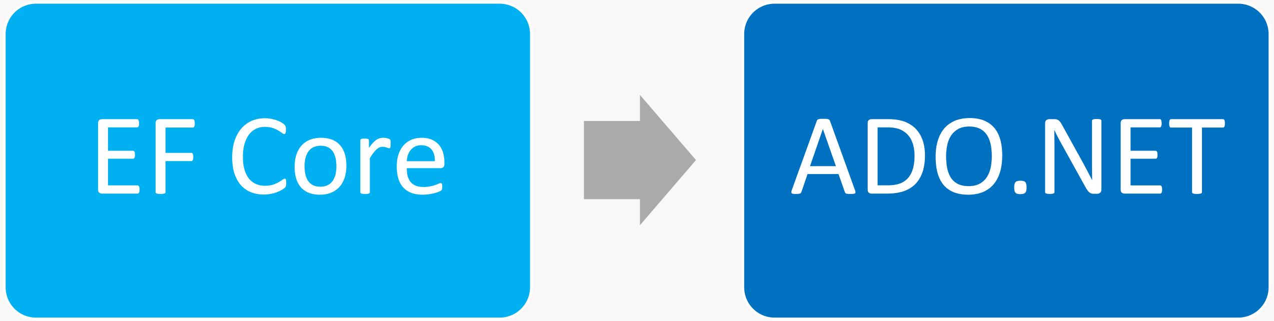
- EF Core is based on ADO.NET
- When **FirstOrDefaultAsync** is called, a DB connection is opened implicitly
- We simply update an entity object in-memory – EF Core‘s change tracking mechanism build a corresponding UPDATE statement when calling **SaveChangesAsync**
- Implicit transactions are in effect for individual calls to the database
- We do not have to dispose of the native resources, this is done by the DI container’s scope

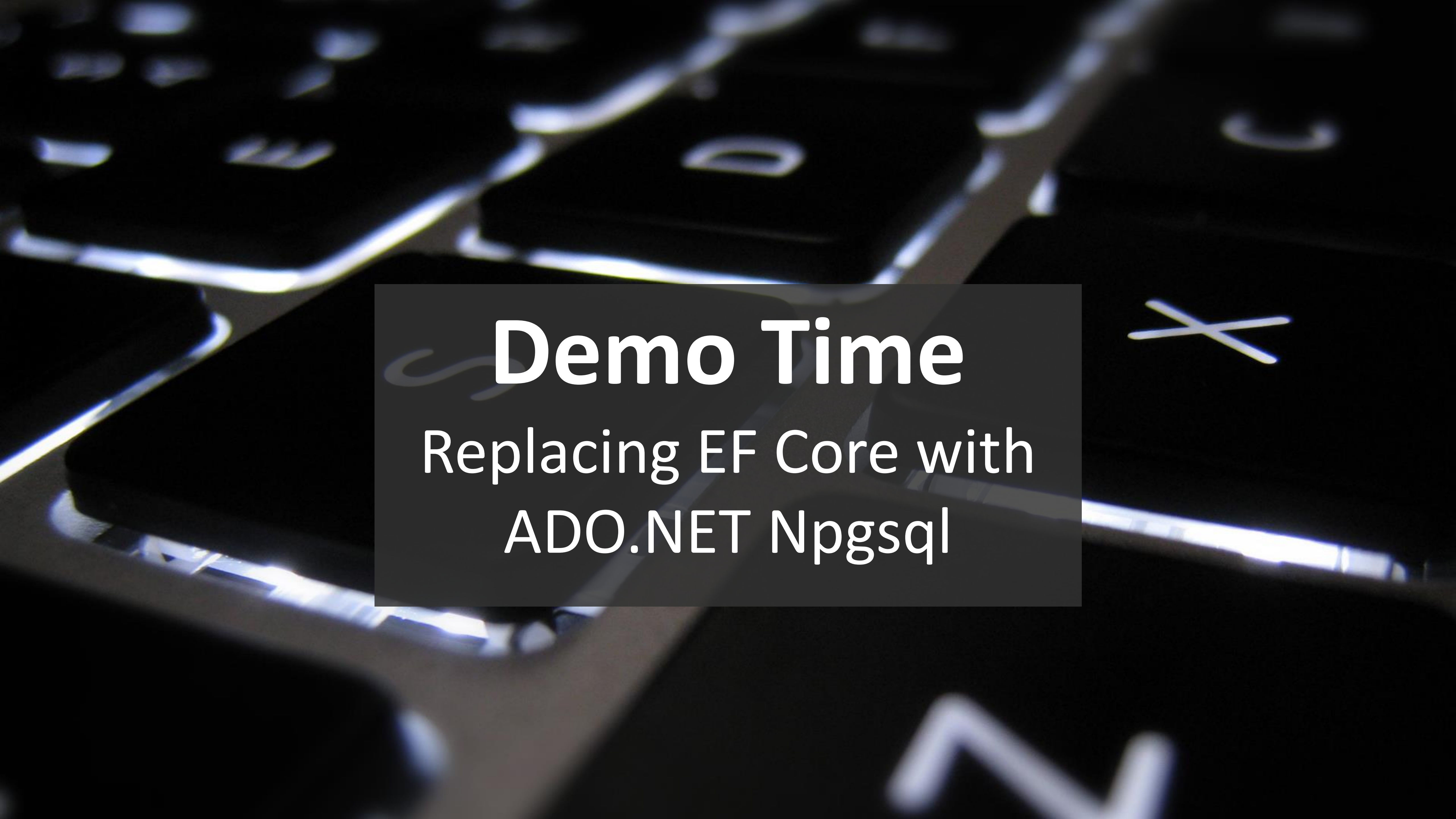


```
public static async Task<IResult> UpdateContact(
    DatabaseContext dbContext,
    UpdateContactDtoValidator validator,
    UpdateContactDto dto,
    CancellationToken cancellationToken = default
)
{
    if (validator.CheckForErrors(dto, out var errors))
    {
        return Results.BadRequest(errors);
    }

    var existingContact = await dbContext.Contacts.FirstOrDefaultAsync(c => c.Id == dto.Id, cancellationToken);
    if (existingContact is null)
    {
        return Results.NotFound();
    }

    existingContact.FirstName = dto.FirstName;
    existingContact.LastName = dto.LastName;
    existingContact.Email = dto.Email;
    existingContact.Phone = dto.PhoneNumber;
    await dbContext.SaveChangesAsync(cancellationToken);

    return Results.NoContent();
}
```

From EF Core to ADO.NET

- We need to remove EF Core's **DbContext** and **DbSet<T>**
- Instead, we need to use **DbConnection**, **DbTransaction**, **DbCommand**, and **DbDataReader**
- We will not re-implement a change-tracking mechanism – this is too complicated and hurts performance
- We will use PostgreSQL and its Npgsql client for .NET





Demo Time

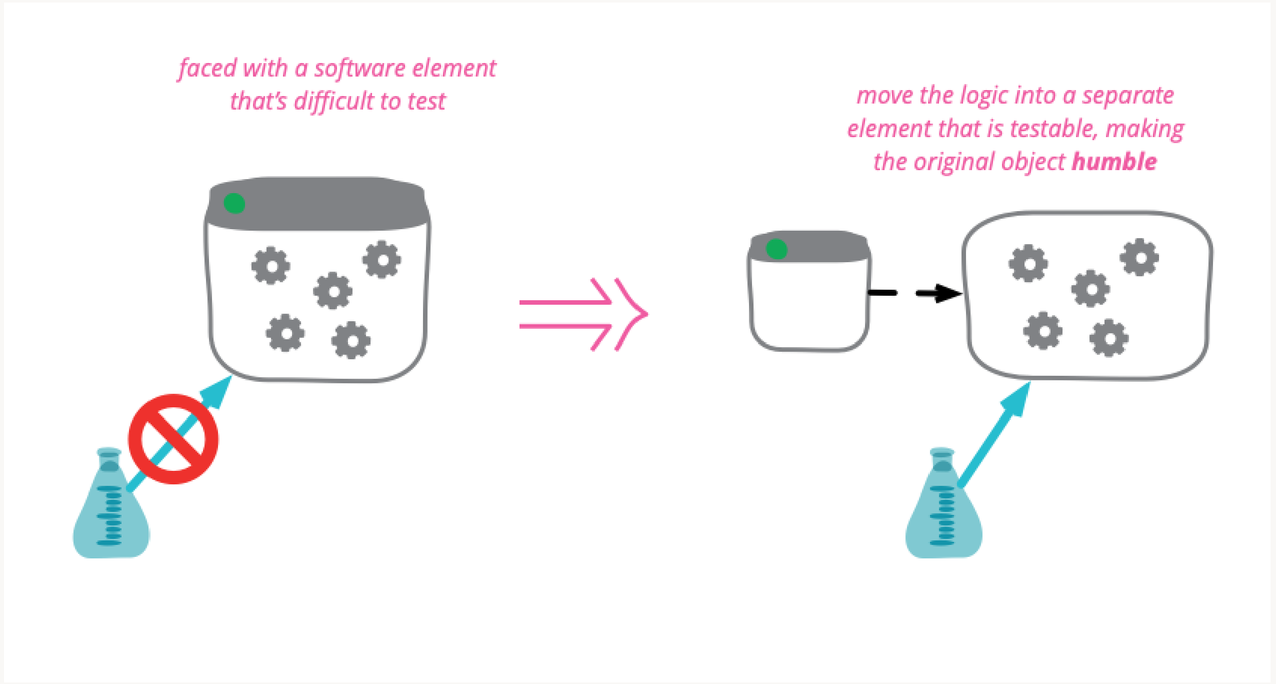
Replacing EF Core with ADO.NET Npgsql

Introducing DB Sessions to abstract Business Logic from Data Access Logic

Tight coupling sucks – Humble Objects to the rescue

“Some program elements are inherently difficult, or even impossible to test. Any logic in these elements is thus prone to bugs and difficult to evolve. To mitigate this problem, move as much as logic as possible out of the hard-to-test element and into other more friendly parts of the code base. By making untestable objects humble, we reduce the chances that they harbor evil bugs.”

From [Martin Fowler's Homepage]

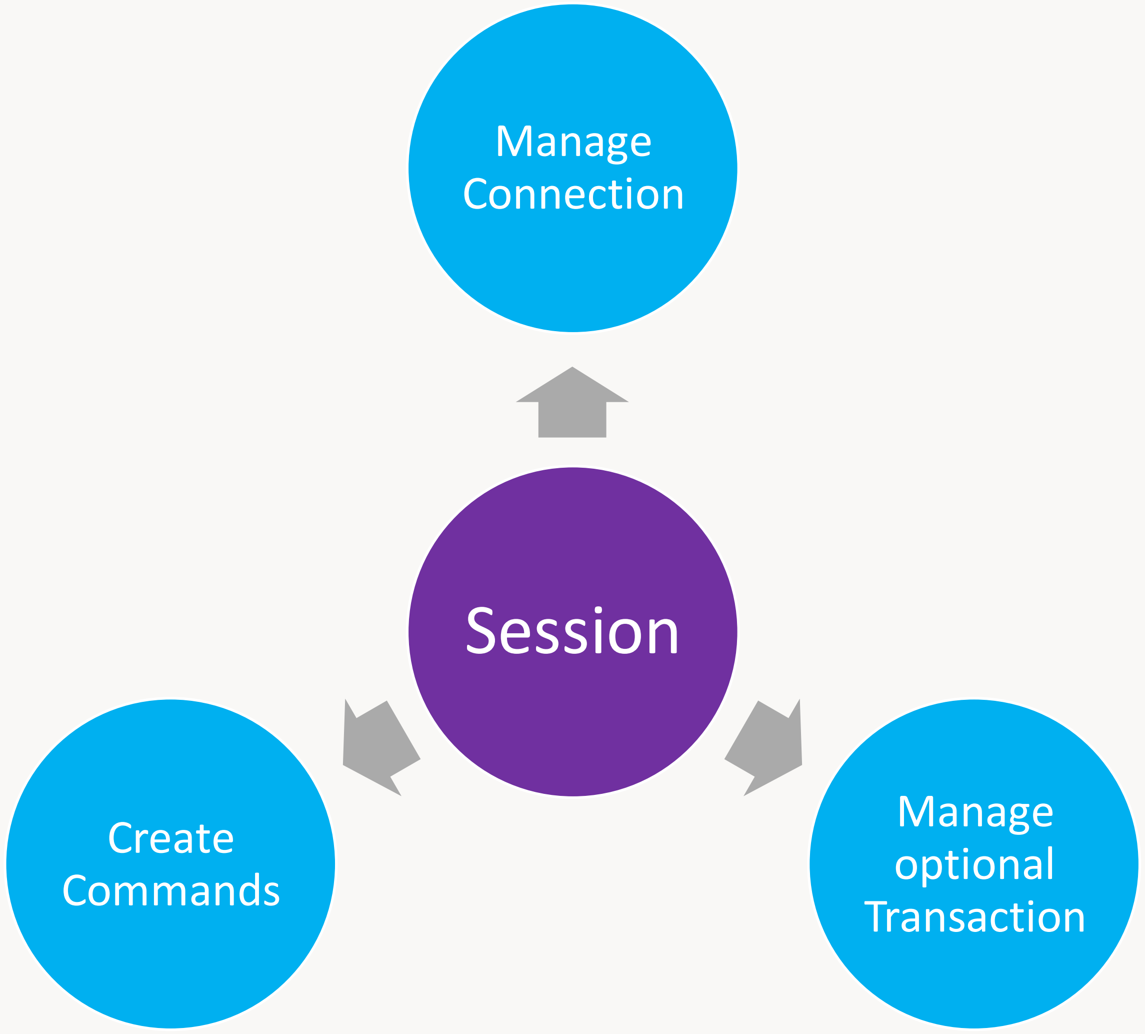

faced with a software element that's difficult to test
move the logic into a separate element that is testable, making the original object **humble**

What should our Humble Objects (DB Sessions) do?

Like **DbContext**, our session should

- manage the lifetime of the DB connection and an optional DB transaction
- Create and execute DB commands
- Optionally receive and deserialize query results via DB Data Reader
- Implement **IDisposable** so that it can be easily integrated with Dependency Injection

We can integrate the initialization logic for connection and transaction into the creation of commands. This avoids opening these resources unnecessarily when no commands are executed against the database.



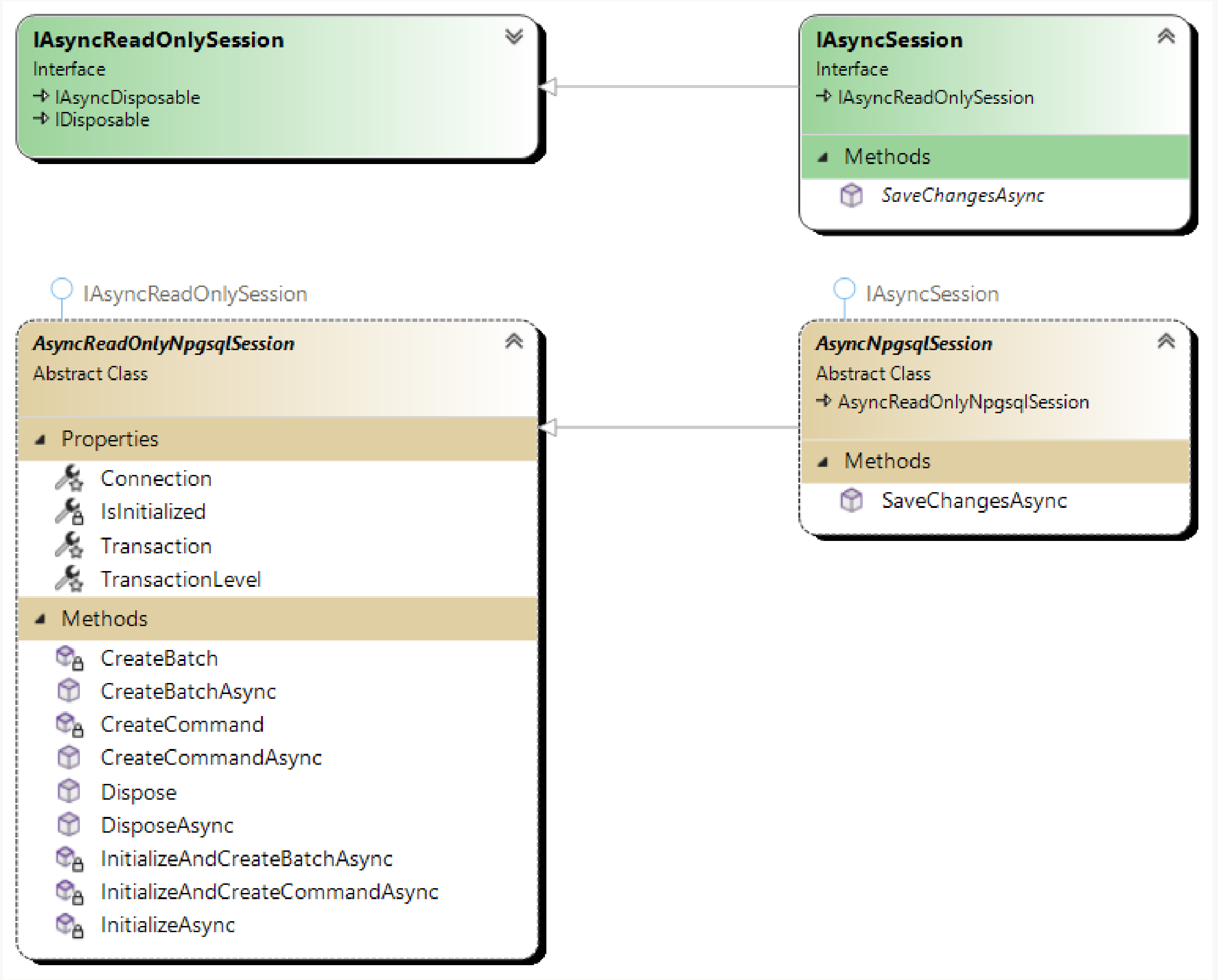
Demo Time

Introducing DB Sessions as Humble Objects

The Session abstractions and Npgsql base classes…

- **IAsyncReadOnlySession** represents the abstraction of a connection to the database where the caller will never commit manipulated data (only for reading data)
- **IAsyncSession** represents the abstraction for a session that can commit changes with the SaveChangesAsync method
- **AsyncReadOnlyNpgsqlSession** and **AsyncNpgsqlSession** are base classes that manage **NpgsqlConnection** and **NpgsqlTransaction**

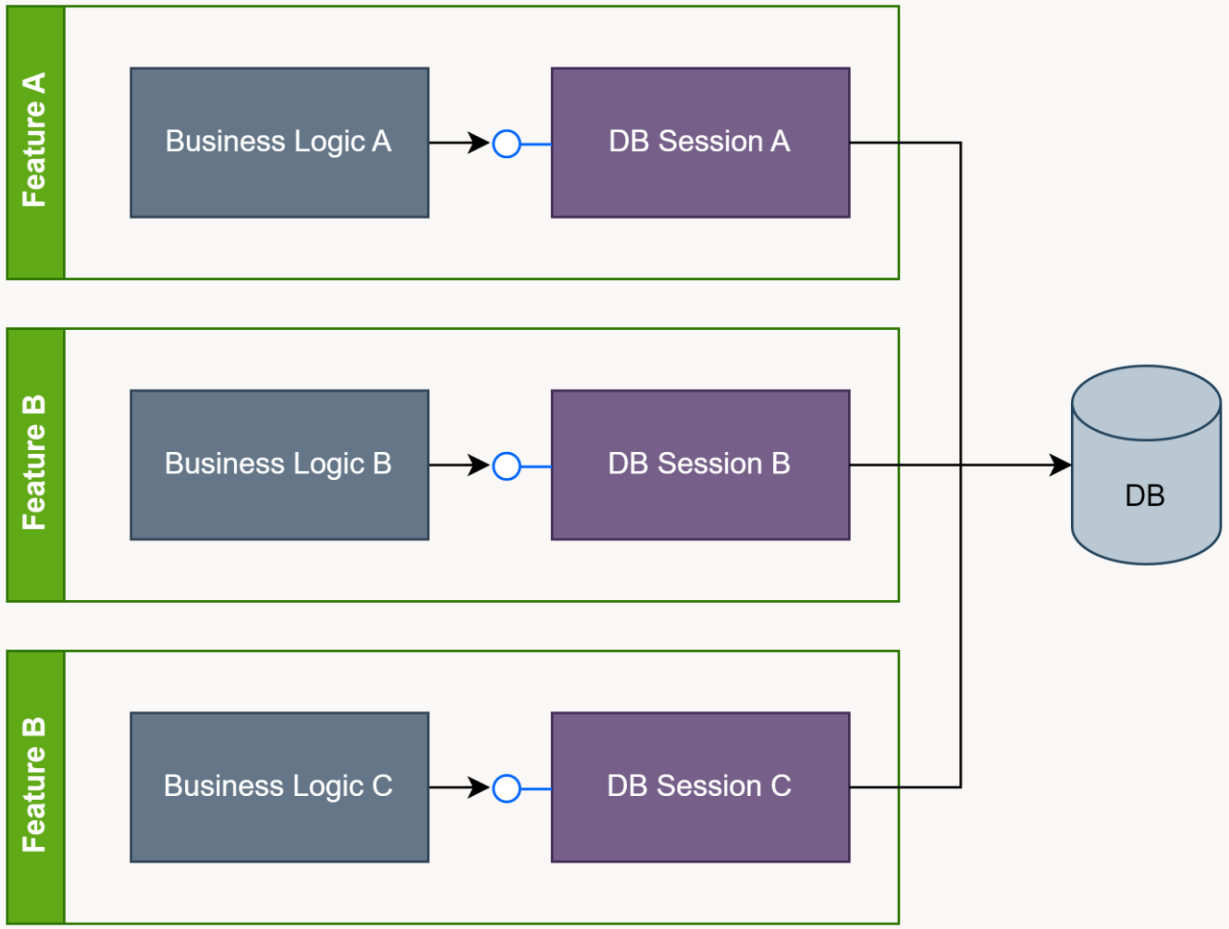
You can also adapt this to other data access technologies.



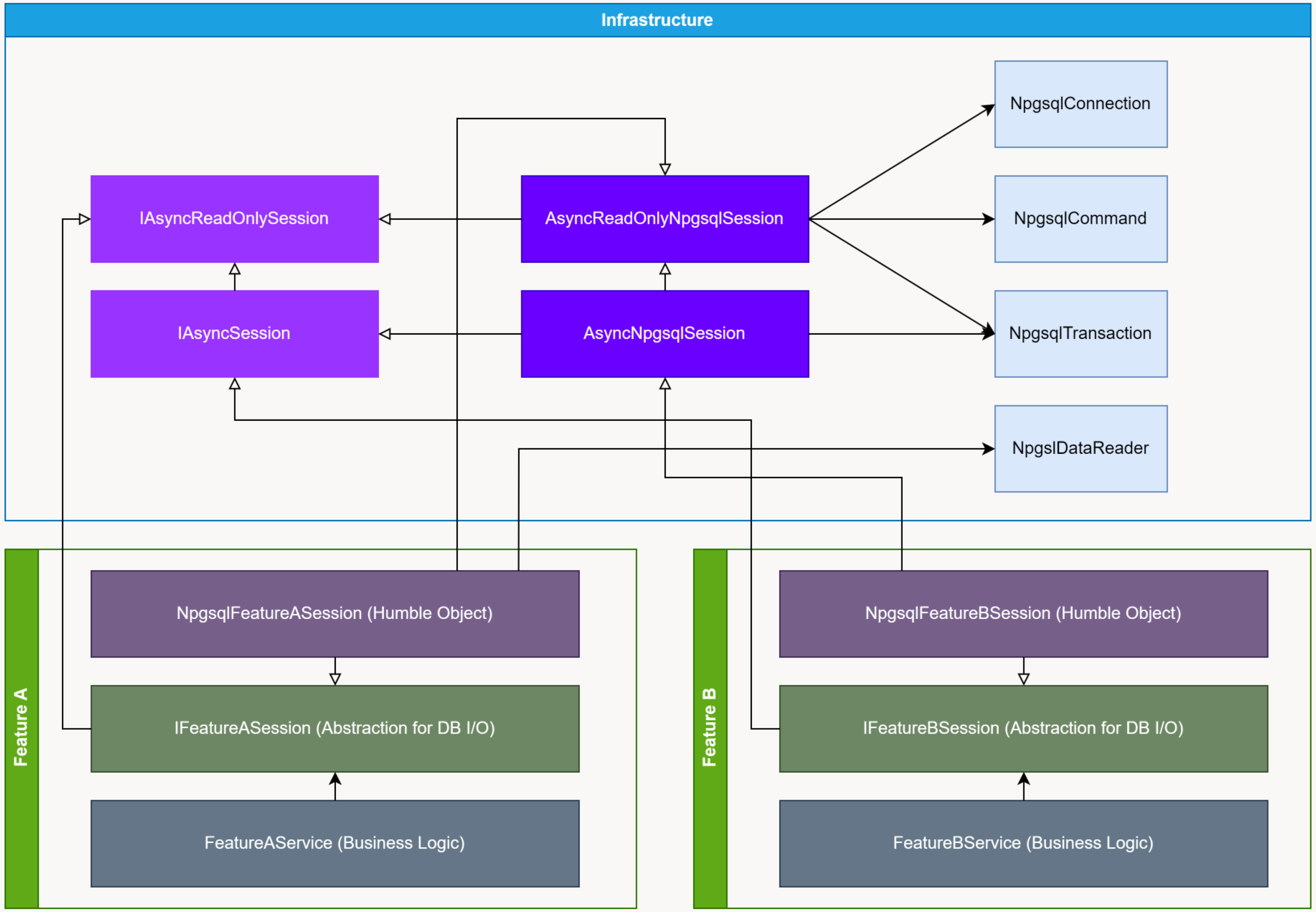
…and how to use them

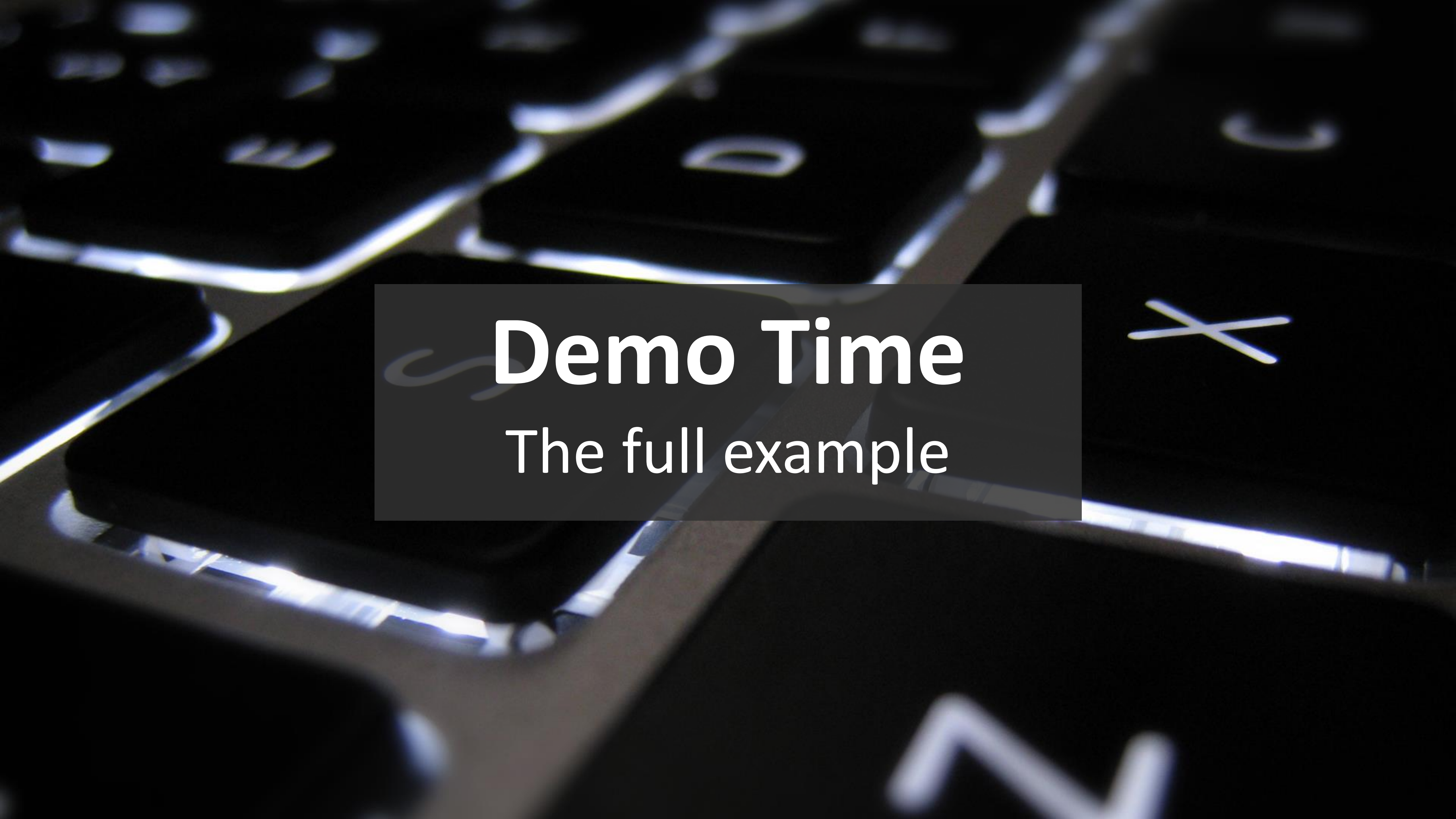
When implementing a new feature that requires a session, do the following:

- Create an abstraction that decouples your in-memory business logic from the I/O calls
- If you need to manipulate data, derive this abstraction from **IAsyncSession**, otherwise use **IAsyncReadOnlySession**
- Add one method per interaction with the DB and the data access library to the interface
- Implement the interface in a session class and perform the corresponding actions in your methods
- In unit tests, exchange the DB Session with a test double – this allows you to go for hard-to-test edge cases
- You can reuse code between session implementations if interactions are equal



Dependencies in detail


Infrastructure
IAsyncReadOnlySession
AsyncReadOnlyNpgsqlSession
IAsyncSession
AsyncNpgsqlSession
NpgsqlConnection
NpgsqlCommand
NpgsqlTransaction
NpgslDataReader
Feature A
NpgsqlFeatureASession (Humble Object)
IFeatureASession (Abstraction for DB I/O)
FeatureAService (Business Logic)
Feature B
NpgsqlFeatureBSession (Humble Object)
IFeatureBSession (Abstraction for DB I/O)
FeatureBService (Business Logic)



Demo Time

The full example

How do DB Sessions relate to Unit of Work and Repository?

Is a Session a Unit of Work?

“Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.”

From Patterns of Enterprise Application Architecture (PEAA)
by Martin Fowler et al.

“The key thing about Unit of Work is that, when it comes time to commit, the Unit of Work decides what to do. It opens a transaction, does any concurrency checking […], and writes changes out to the database. Application programmers never explicitly call methods for database updates. This way they don’t have to keep track of what’s changed or worry about how referential integrity affects the order in which they need to do things.”

Strictly speaking, a Session is not a Unit of Work

A Session handles the DB Connection and the transaction(s) similarly to how a Unit of Work would do it. **But the implementation of SaveChangesAsync does not use Change Tracking to determine INSERT, UPDATE, and DELETE statements and executes them in the right order (we simply commit a transaction).**

This is a key requirement according to PEAA – this means **there is no Micro ORM that implements the Unit of Work pattern fully.**

Most Micro ORMs make one hop to the database per DB Command – there is no packaging of several statements into one DB Command to reduce I/O calls.

If our implementation would adapt EF Core, NHibernate or another “full” ORM, our sessions would be Units of Work.

Is a Session a Repository?

“Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.”

From Patterns of Enterprise Application Architecture (PEAA)
by Martin Fowler et al.

“A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added and removed from the Repository as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.”

A Session is not a Repository

- A Session does not behave like an in-memory collection
- It does not encapsulate a single object type
- Sessions do not allow arbitrary querying (via **IQueryable<T>**)

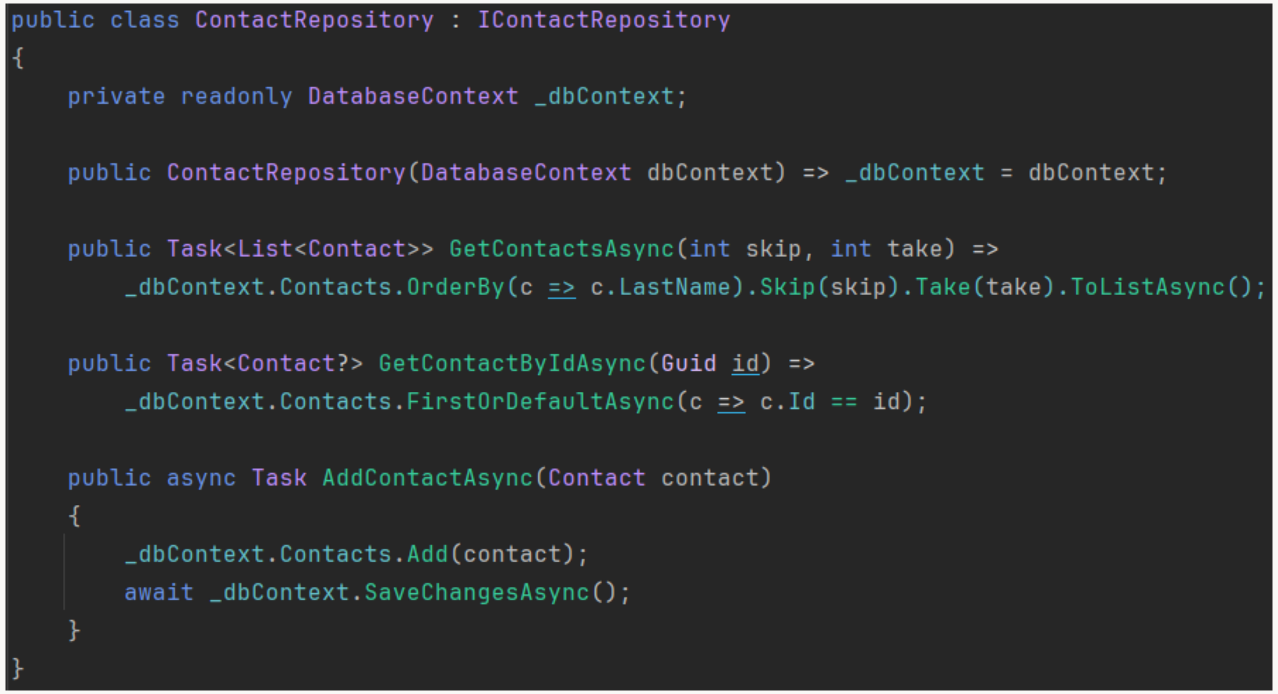
EF Core‘s **DbContext** is an implementation of the Unit of Work pattern.

EF Core‘s **DbSet<T>** is an implementation of the Repository pattern.

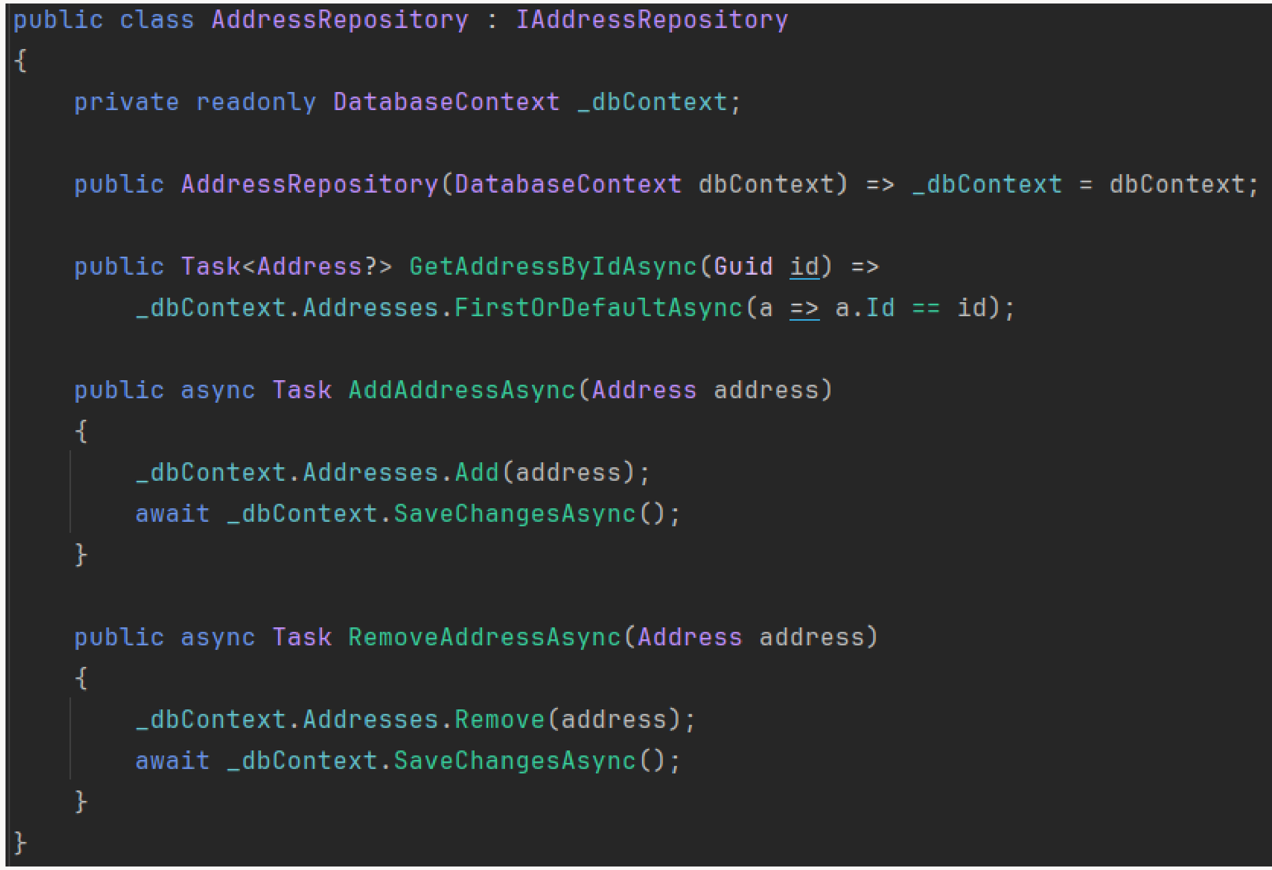
Your Business Logic should not program against a Repository!

Odd implementations of Repository and Unit of Work

The Reposi-Unit – Code



```
public class ContactRepository : IContactRepository
{
    private readonly DatabaseContext _dbContext;

    public ContactRepository(DatabaseContext dbContext) => _dbContext = dbContext;

    public Task<List<Contact>> GetContactsAsync(int skip, int take) =>
        _dbContext.Contacts.OrderBy(c => c.LastName).Skip(skip).Take(take).ToListAsync();

    public Task<Contact?> GetContactByIdAsync(Guid id) =>
        _dbContext.Contacts.FirstOrDefaultAsync(c => c.Id == id);

    public async Task AddContactAsync(Contact contact)
    {
        _dbContext.Contacts.Add(contact);
        await _dbContext.SaveChangesAsync();
    }
}
```



```
public class AddressRepository : IAddressRepository
{
    private readonly DatabaseContext _dbContext;

    public AddressRepository(DatabaseContext dbContext) => _dbContext = dbContext;

    public Task<Address?> GetAddressByIdAsync(Guid id) =>
        _dbContext.Addresses.FirstOrDefaultAsync(a => a.Id == id);

    public async Task AddAddressAsync(Address address)
    {
        _dbContext.Addresses.Add(address);
        await _dbContext.SaveChangesAsync();
    }

    public async Task RemoveAddressAsync(Address address)
    {
        _dbContext.Addresses.Remove(address);
        await _dbContext.SaveChangesAsync();
    }
}
```

The Reposi-Unit – Description

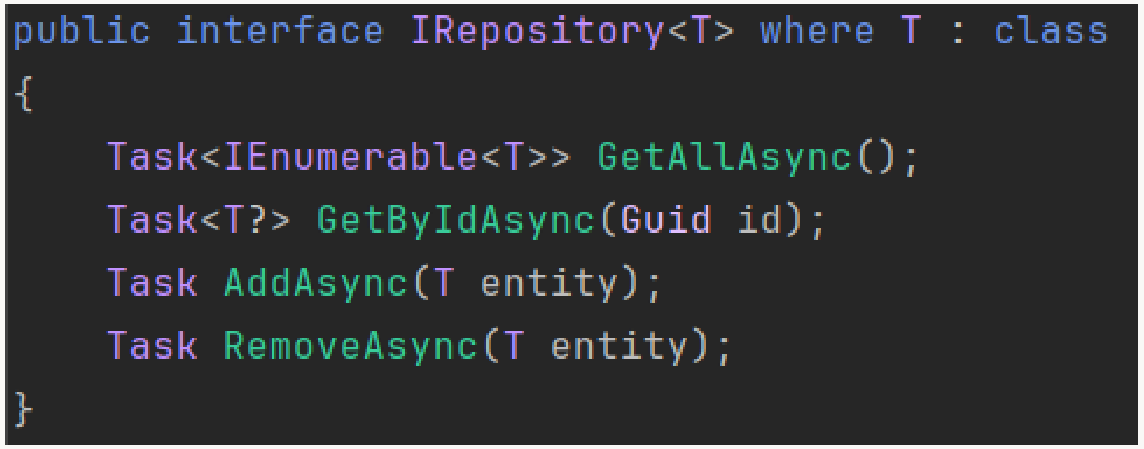
- This “Repository” encapsulates one entity type (typically one table, can be one Aggregate Root when implementing Domain-Driven Design)
- Each interaction with the entity will be one method and is available to all clients
- It looks like a Unit of Work but does not offer a **SaveChangesAsync** method to the caller – instead, it calls **SaveChangesAsync** implicitly when adding or removing entities (Business Logic should not care about when to commit a transaction)
- Business Logic that interacts with several tables will reference the corresponding repositories and call some of the methods on them

The Reposi-Unit – Criticism

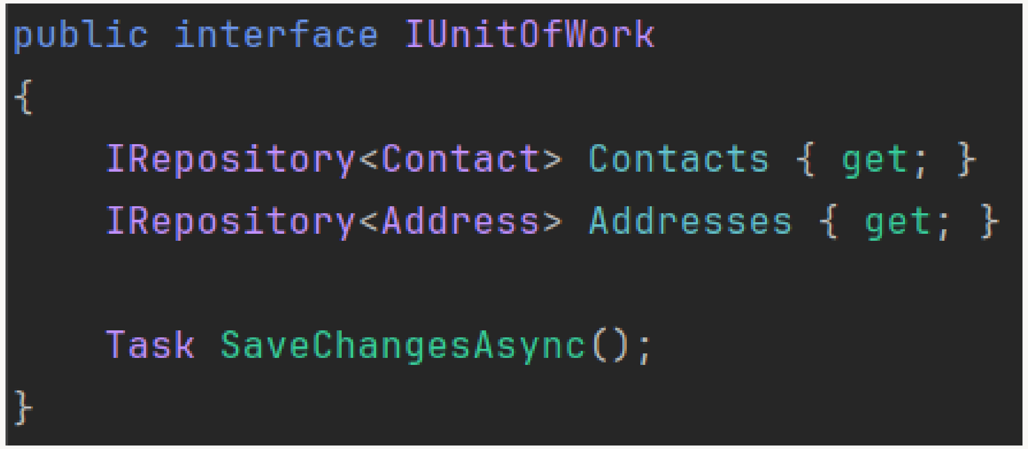
- This pattern violates the Dependency Inversion Principle and the Interface Segregation Principle. Callers see too many methods. The interface is not designed to fit for the client, but for the implementation. This makes it harder to implement test doubles in Unit Tests.
- When interacting with several repositories (or sometimes just even one) within the same service, **SaveChangesAsync** will be called several times. This most likely breaks the transactional assurances the database gives us: what happens if the first **SaveChangesAsync** call succeeds, but a subsequent one does not?

In my opinion, this is an Anti-Pattern and only works in the simple (CRUD) scenarios where we interact with only one table. Microsoft discuss this pattern in their docs, but they at least say it “shouldn’t be mandatory”.

Generic Repository and Unit of Work – Code



```
public interface IRepository<T> where T : class
{
    Task<IEnumerable<T>> GetAllAsync();
    Task<T?> GetByIdAsync(Guid id);
    Task AddAsync(T entity);
    Task RemoveAsync(T entity);
}
```



```
public interface IUnitOfWork
{
    IRepository<Contact> Contacts { get; }
    IRepository<Address> Addresses { get; }

    Task SaveChangesAsync();
}
```

Generic Repository and Unit of Work – Description

- The generic repository provides dedicated methods that each repository must have
- Additional required methods for an entity type can be added by deriving an interface and subclassing the implementation of a generic repository
- Sometimes, the returned types are not the entity classes, but extra DTOs that the repository maps to and from
- All repositories are listed in a Unit of Work – this is injected into each part of the business logic that requires database interaction

Generic Repository and Unit of Work – Criticism

- Again, this is a violation of the Dependency Inversion Principle and the Interface Segregation Principle. Callers will see repositories on the Unit of Work and members on the repositories that they will not call.
- The generic repositories make it hard to load data from several tables. Developers use one repository to load all data from one table, the next repository to load all data from another table, and then combine them in memory. This does not scale.

In conclusion, Generic Repository and Unit of Work are bad abstractions. It is not designed for the caller, and it hinders you to effectively use the capabilities of the database except for transactions.

Which ORM's can we use with .NET Native AOT?

.NET Object/Relational Mappers and Native AOT

Tested:

- Dapper AOT ✓ ⚠️
- LinqToDB ✓ ⚠️
- Entity Framework Core ❌
- NHibernate ❌
- ServiceStack.OrmLite ❌
- PetaPoco ❌
- RepoDB ❌

Not tested:

- DevExpress XPO ❔
- LLBLGenPro ❔
- Massive ❔
- SqlMarshal ❔

Demo Time

Dapper AOT

Dapper AOT – It works, but…

- Currently, only the extension methods of Dapper are intercepted and replaced with Source Generator code
- There is no support for **CommandDefinition** and cancellation tokens
- There is no support for gathering several statements into one command (except for string concatenation)

When targeting PostgreSQL, I prefer to simply work with Npgsql instead of Dapper – but this situation should be evaluated individually for each ADO.NET provider.

Conclusion

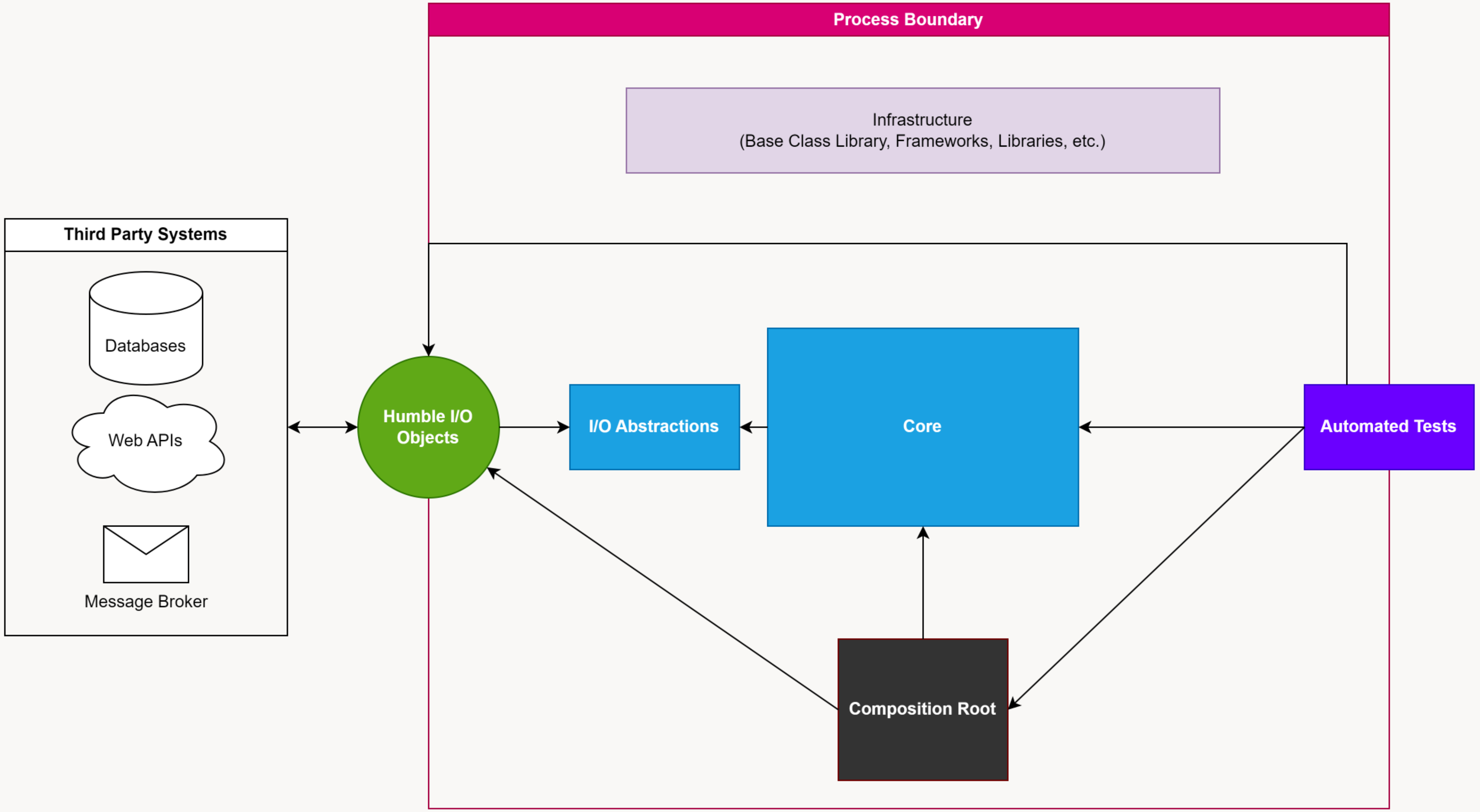
Data Access in Native AOT

- Many ORMs do not work with .NET Native AOT due to unbound reflection and/or runtime code generation
- Your best bet today: ADO.NET (with or without Dapper AOT)
- We learned that ADO.NET code can be structured in a similar fashion to EF Core by programming against an abstract session (Unit of Work?)
- In fact, we can abstract any data access technology with this pattern
- Take #1: do not program against repositories in your Business Logic – they should be an implementation detail of the Data Access Layer
- Take #2: do not hide the SaveChangesAsync call from your Business Logic – it should decide when to commit, but not how a commit is performed

The final comparison: Session vs. Unit of Work

Feature	Session	Unit of Work
DB Connection Management	✅	✅
DB Transaction Management	✅	✅
Change Tracking	⚠️ Depends on the underlying data access technology. Npgsql allows to easily gather several statements into one command (without string concatenation and the possible SQL injection attack)	✅

Humble Objects and their impact on design



Sources

- XUnit Test Patterns – Gerard Meszaros
- Patterns of Enterprise Application Architecture – Martin Fowler et al.
- Domain-Driven Design: Tackling Complexity at the Heart of Software – Eric Evans
- Design the infrastructure persistence layer – Microsoft Learn
- Agile Principles, Patterns, and Practices in C# - Robert C. Martin, Micah Martin
- NativeAOT support – GitHub dotnet efcore issue
- Dapper AOT – GitHub

Danke schön!

Demos und Slides:
https://github.com/thinktecture-labs/dotnet-native-aot-webinar

https://www.thinktecture.com/wissen/
https://labs.thinktecture.com/

https://www.thinktecture.com/ueber-uns/karriere/

Werde Teil des Teams

Angular Developer mit UX/UI-Fokus (m/w/d)
ZUR STELLENBESCHREIBUNG >

.NET Developer mit Cloud-Fokus (m/w/d)
ZUR STELLENBESCHREIBUNG >

Angular Developer (m/w/d)
ZUR STELLENBESCHREIBUNG >

Kenny Pflug
https://thinktecture.com/kenny-pflug