DbContext Lifetime, Configuration, and Initialization

11/07/2020 • 11 minutes to read • 

In this article

This article shows basic patterns for initialization and configuration of a DbContext instance.

The DbContext lifetime

The lifetime of a `DbContext` begins when the instance is created and ends when the instance is disposed. A `DbContext` instance is designed to be used for a *single* unit-of-work. This means that the lifetime of a `DbContext` instance is usually very short.

> **Tip**
>
> To quote Martin Fowler from the link above, "A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work."

A typical unit-of-work when using Entity Framework Core (EF Core) involves:

- Creation of a `DbContext` instance
- Tracking of entity instances by the context. Entities become tracked by
  - Being returned from a query

  - Being added or attached to the context
- Changes are made to the tracked entities as needed to implement the business rule
- SaveChanges or SaveChangesAsync is called. EF Core detects the changes made and writes them to the database.
- The `DbContext` instance is disposed

> **Important**
>
> - It is very important to dispose the **DbContext** after use. This ensures both that any unmanaged resources are freed, and that any events or other hooks are unregistered so as to prevent memory leaks in case the instance remains referenced.
> - **DbContext is not thread-safe**. Do not share contexts between threads. Make sure to **await** all async calls before continuing to use the context instance.
> - An **InvalidOperationException** thrown by EF Core code can put the context into an unrecoverable state. Such exceptions indicate a program error and are not designed to be recovered from.

DbContext in dependency injection for ASP.NET Core

In many web applications, each HTTP request corresponds to a single unit-of-work. This makes tying the context lifetime to that of the request a good default for web applications.

ASP.NET Core applications are configured using dependency injection. EF Core can be added to this configuration using AddDbContext in the ConfigureServices method of `Startup.cs`. For example:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDbContext<ApplicationDbContext>(
        options =>
options.UseSqlServer("name=ConnectionStrings:DefaultConnection"));
}
```

This example registers a `DbContext` subclass called `ApplicationDbContext` as a scoped service in the ASP.NET Core application service provider (a.k.a. the dependency injection container). The context is configured to use the SQL Server database provider and will read the connection string from ASP.NET Core configuration. It typically does not matter *where* in `ConfigureServices` the call to `AddDbContext` is made.

The `ApplicationDbContext` class must expose a public constructor with a `DbContextOptions<ApplicationDbContext>` parameter. This is how context configuration from `AddDbContext` is passed to the `DbContext`. For example:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

`ApplicationDbContext` can then be used in ASP.NET Core controllers or other services through constructor injection. For example:

C# Copy

```
public class MyController
{
    private readonly ApplicationDbContext _context;

    public MyController(ApplicationDbContext context)
    {
        _context = context;
    }
}
```

The final result is an `ApplicationDbContext` instance created for each request and passed to the controller to perform a unit-of-work before being disposed when the request ends.

Read further in this article to learn more about configuration options. In addition, see App startup in ASP.NET Core and Dependency injection in ASP.NET Core for more information on configuration and dependency injection in ASP.NET Core.

Simple DbContext initialization with 'new'

`DbContext` instances can be constructed in the normal .NET way, for example with `new` in C#. Configuration can be performed by overriding the `OnConfiguring` method, or by passing options to the constructor. For example:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=Test");
    }
}
```

This pattern also makes it easy to pass configuration like the connection string via the `DbContext` constructor. For example:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    private readonly string _connectionString;

    public ApplicationDbContext(string connectionString)
    {
        _connectionString = connectionString;
    }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connectionString);
    }
}
```

Alternately, `DbContextOptionsBuilder` can be used to create a `DbContextOptions` object that is then passed to the `DbContext` constructor. This allows a `DbContext` configured for dependency injection to also be constructed explicitly. For example, when using `ApplicationDbContext` defined for ASP.NET Core web apps above:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

The `DbContextOptions` can be created and the constructor can be called explicitly:

C# Copy

```
var contextOptions = new DbContextOptionsBuilder<ApplicationDbContext>()
    .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test")
    .Options;

using var context = new ApplicationDbContext(contextOptions);
```

Using a DbContext factory (e.g. for Blazor)

Some application types (e.g. ASP.NET Core Blazor) use dependency injection but do not create a service scope that aligns with the desired `DbContext` lifetime. Even where such an alignment does exist, the application may need to perform multiple units-of-work within this scope. For example, multiple units-of-work within a single HTTP request.

In these cases, AddDbContextFactory can be used to register a factory for creation of `DbContext` instances. For example:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContextFactory<ApplicationDbContext>(options =>
        options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test"));
}
```

The `ApplicationDbContext` class must expose a public constructor with a `DbContextOptions<ApplicationDbContext>` parameter. This is the same pattern as used in the traditional ASP.NET Core section above.

C# Copy

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

The `DbContextFactory` factory can then be used in other services through constructor injection. For example:

C# Copy

```
private readonly IDbContextFactory<ApplicationDbContext> _contextFactory;

public MyController(IDbContextFactory<ApplicationDbContext> contextFactory)
{
    _contextFactory = contextFactory;
}
```

The injected factory can then be used to construct DbContext instances in the service code. For example:

C# Copy

```
public void DoSomething()
{
    using (var context = _contextFactory.CreateDbContext())
    {
        // ...
    }
}
```

Notice that the `DbContext` instances created in this way are **not** managed by the application's service provider and therefore must be disposed by the application.

See ASP.NET Core Blazor Server with Entity Framework Core for more information on using EF Core with Blazor.

DbContextOptions

The starting point for all `DbContext` configuration is DbContextOptionsBuilder. There are three ways to get this builder:

- In `AddDbContext` and related methods
- In `OnConfiguring`
- Constructed explicitly with `new`

Examples of each of these are shown in the preceding sections. The same configuration can be applied regardless of where the builder comes from. In addition, `OnConfiguring` is always called regardless of how the context is constructed. This means `OnConfiguring` can be used to perform additional configuration even when `AddDbContext` is being used.

Configuring the database provider

Each `DbContext` instance must be configured to use one and only one database provider. (Different instances of a `DbContext` subtype can be used with different database providers, but a single instance must only use one.) A database provider is configured using a specific `Use*` " call. For example, to use the SQL Server database provider:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=Test");
    }
}
```

These `Use*` " methods are extension methods implemented by the database provider. This means that the database provider NuGet package must be installed before the extension method can be used.

> **Tip**
>
> EF Core database providers make extensive use of **extension methods**. If the compiler indicates that a method cannot be found, then make sure that the provider's NuGet package is installed and that you have `using Microsoft.EntityFrameworkCore;` in your code.

The following table contains examples for common database providers.

Database system	Example configuration	NuGet package
SQL Server or Azure SQL	`.UseSqlServer(connectionString)`	Microsoft.EntityFrameworkCore.SqlSe
Azure Cosmos DB	`.UseCosmos(connectionString, databaseName)`	Microsoft.EntityFrameworkCore.Cosm
SQLite	`.UseSqlite(connectionString)`	Microsoft.EntityFrameworkCore.Sqlite
EF Core in-memory database	`.UseInMemoryDatabase(databaseName)`	Microsoft.EntityFrameworkCore.InMe
PostgreSQL*	`.UseNpgsql(connectionString)`	Npgsql.EntityFrameworkCore.Postgre
MySQL/MariaDB*	`.UseMySql((connectionString)`	Pomelo.EntityFrameworkCore.MySql
Oracle*	`.UseOracle(connectionString)`	Oracle.EntityFrameworkCore

*These database providers are not shipped by Microsoft. See Database Providers for more information about database providers.

> **⚠ Warning**
>
> The EF Core in-memory database is not designed for production use. In addition, it may not be the best choice even for testing. See **Testing Code That Uses EF Core** for more information.

See Connection Strings for more information on using connection strings with EF Core.

Optional configuration specific to the database provider is performed in an additional provider-specific builder. For example, using EnableRetryOnFailure to configure retries for connection resiliency when connecting to Azure SQL:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(
                @"Server=(localdb)\mssqllocaldb;Database=Test",
                providerOptions =>
                    {
                        providerOptions.EnableRetryOnFailure();
                    });
    }
}
```

Tip

The same database provider is used for SQL Server and Azure SQL. However, it is recommended that **connection resiliency** be used when connecting to SQL Azure.

See Database Providers for more information on provider-specific configuration.

Other DbContext configuration

Other `DbContext` configuration can be chained either before or after (it makes no difference which) the `Use*` call. For example, to turn on sensitive-data logging:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder
            .EnableSensitiveDataLogging()
            .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Test");
    }
}
```

The following table contains examples of common methods called on `DbContextOptionsBuilder`.

DbContextOptionsBuilder method	What it does	Learn more
UseQueryTrackingBehavior	Sets the default tracking behavior for queries	Query Tracking Behavior
LogTo	A simple way to get EF Core logs (EF Core 5.0 and later)	Logging, Events, and Diagnostics
UseLoggerFactory	Registers an `Microsoft.Extensions.Logging` factory	Logging, Events, and Diagnostics
EnableSensitiveDataLogging	Includes application data in exceptions and logging	Logging, Events, and Diagnostics
EnableDetailedErrors	More detailed query errors (at the expense of performance)	Logging, Events, and Diagnostics
ConfigureWarnings	Ignore or throw for warnings and other events	Logging, Events, and Diagnostics
AddInterceptors	Registers EF Core interceptors	Logging, Events, and Diagnostics
UseLazyLoadingProxies	Use dynamic proxies for lazy-loading	Lazy Loading
UseChangeTrackingProxies	Use dynamic proxies for change-tracking	Coming soon...

> **Note**
>
> **UseLazyLoadingProxies** and **UseChangeTrackingProxies** are extension methods from the **Microsoft.EntityFrameworkCore.Proxies** NuGet package. This kind of ".UseSomething()" call is the recommended way to configure and/or use EF Core extensions contained in other packages.

`DbContextOptions` verses `DbContextOptions<TContext>`

Most `DbContext` subclasses that accept a `DbContextOptions` should use the generic `DbContextOptions<TContext>` variation. For example:

C# Copy

```
public sealed class SealedApplicationDbContext : DbContext
{
    public
SealedApplicationDbContext(DbContextOptions<SealedApplicationDbContext>
contextOptions)
        : base(contextOptions)
    {
    }
}
```

This ensures that the correct options for the specific `DbContext` subtype are resolved from dependency injection, even when multiple `DbContext` subtypes are registered.

> **Tip**
>
> Your DbContext does not need to be sealed, but sealing is best practice to do so for classes not designed to be inherited from.

However, if the `DbContext` subtype is itself intended to be inherited from, then it should expose a protected constructor taking a non-generic `DbContextOptions`. For example:

C# Copy

```
public abstract class ApplicationDbContextBase : DbContext
{
    protected ApplicationDbContextBase(DbContextOptions contextOptions)
        : base(contextOptions)
    {
    }
}
```

This allows multiple concrete subclasses to call this base constructor using their different generic `DbContextOptions<TContext>` instances. For example:

C# Copy

```
public sealed class ApplicationDbContext1 : ApplicationDbContextBase
{
    public ApplicationDbContext1(DbContextOptions<ApplicationDbContext1>
contextOptions)
        : base(contextOptions)
    {
```

```
    }
}

public sealed class ApplicationDbContext2 : ApplicationDbContextBase
{
    public ApplicationDbContext2(DbContextOptions<ApplicationDbContext2>
contextOptions)
        : base(contextOptions)
    {
    }
}
```

Notice that this is exactly the same pattern as when inheriting from `DbContext` directly. That is, the `DbContext` constructor itself accepts a non-generic `DbContextOptions` for this reason.

A `DbContext` subclass intended to be both instantiated and inherited from should expose both forms of constructor. For example:

C# Copy

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
contextOptions)
        : base(contextOptions)
    {
    }

    protected ApplicationDbContext(DbContextOptions contextOptions)
        : base(contextOptions)
    {
    }
}
```

Design-time DbContext configuration

EF Core design-time tools such as those for EF Core migrations need to be able to discover and create a working instance of a `DbContext` type in order to gather details about the application's entity types and how they map to a database schema. This process can be automatic as long as the tool can easily create the `DbContext` in such a way that it will be configured similarly to how it would be configured at run-time.

While any pattern that provides the necessary configuration information to the `DbContext` can work at run-time, tools that require using a `DbContext` at design-time can only work with a limited number of patterns. These are covered in more detail in Design-Time Context Creation.

Avoiding DbContext threading issues

Entity Framework Core does not support multiple parallel operations being run on the same `DbContext` instance. This includes both parallel execution of async queries and any explicit concurrent use from multiple threads. Therefore, always `await` async calls immediately, or use separate `DbContext` instances for operations that execute in parallel.

When EF Core detects an attempt to use a `DbContext` instance concurrently, you'll see an `InvalidOperationException` with a message like this:

> A second operation started on this context before a previous operation completed. This is usually caused by different threads using the same instance of DbContext, however instance members are not guaranteed to be thread safe.

When concurrent access goes undetected, it can result in undefined behavior, application crashes and data corruption.

There are common mistakes that can inadvertently cause concurrent access on the same `DbContext` instance:

Asynchronous operation pitfalls

Asynchronous methods enable EF Core to initiate operations that access the database in a non-blocking way. But if a caller does not await the completion of one of these methods, and proceeds to perform other operations on the `DbContext`, the state of the `DbContext` can be, (and very likely will be) corrupted.

Always await EF Core asynchronous methods immediately.

Implicitly sharing DbContext instances via dependency injection

The AddDbContext extension method registers `DbContext` types with a scoped lifetime by default.

This is safe from concurrent access issues in most ASP.NET Core applications because there is only one thread executing each client request at a given time, and because each request gets a separate dependency injection scope (and therefore a separate `DbContext` instance). For Blazor Server hosting model, one logical request is used for maintaining the Blazor user circuit, and thus only one scoped DbContext instance is available per user circuit if the default injection scope is used.

Any code that explicitly executes multiple threads in parallel should ensure that `DbContext` instances aren't ever accessed concurrently.

Using dependency injection, this can be achieved by either registering the context as scoped, and creating scopes (using `IServiceScopeFactory`) for each thread, or by registering the `DbContext` as transient (using the overload of `AddDbContext` which takes a `ServiceLifetime` parameter).

More reading

- Read Dependency Injection to learn more about using DI.
- Read Testing for more information.