Migrations Overview

Article • 01/12/2023

In real world projects, data models change as features get implemented: new entities or properties are added and removed, and database schemas need to be changed accordingly to be kept in sync with the application. The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

At a high level, migrations function in the following way:

- When a data model change is introduced, the developer uses EF Core tools to add a corresponding migration describing the updates necessary to keep the database schema in sync. EF Core compares the current model against a snapshot of the old model to determine the differences, and generates migration source files; the files can be tracked in your project's source control like any other source file.
- Once a new migration has been generated, it can be applied to a database in various ways. EF Core records all applied migrations in a special history table, allowing it to know which migrations have been applied and which haven't.

The rest of this page is a step-by-step beginner's guide for using migrations. Consult the other pages in this section for more in-depth information.

Getting started

Let's assume you've just completed your first EF Core application, which contains the following simple model:

C#

```csharp
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

During development, you may have used the Create and Drop APIs to iterate quickly, changing your model as needed; but now that your application is going to production, you need a way to safely evolve the schema without dropping the entire database.

Install the tools

First, you'll have to install the EF Core command-line tools:

- We generally recommend using the .NET Core CLI tools, which work on all platforms.
- If you're more comfortable working inside Visual Studio or have experience with EF6 migrations, you can also use the Package Manager Console tools.

Create your first migration

You're now ready to add your first migration! Instruct EF Core to create a migration named **InitialCreate**:

.NET Core CLI

.NET CLI

```
dotnet ef migrations add InitialCreate
```

EF Core will create a directory called **Migrations** in your project, and generate some files. It's a good idea to inspect what exactly EF Core generated - and possibly amend it - but we'll skip over that for now.

Create your database and schema

At this point you can have EF create your database and create your schema from the migration. This can be done via the following:

.NET Core CLI

.NET CLI

```
dotnet ef database update
```

That's all there is to it - your application is ready to run on your new database, and you didn't need to write a single line of SQL. Note that this way of applying migrations is ideal for local development, but is less suitable for production environments - see the Applying Migrations page for more info.

Evolving your model

A few days have passed, and you're asked to add a creation timestamp to your blogs. You've done the necessary changes to your application, and your model now looks like this:

C#

```csharp
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime CreatedTimestamp { get; set; }
}
```

Your model and your production database are now out of sync - we must add a new column to your database schema. Let's create a new migration for this:

.NET Core CLI

.NET CLI

```
dotnet ef migrations add AddBlogCreatedTimestamp
```

Note that we give migrations a descriptive name, to make it easier to understand the project history later.

Since this isn't the project's first migration, EF Core now compares your updated model against a snapshot of the old model, before the column was added; the model snapshot is one of the files generated by EF Core when you add a migration, and is checked into source control. Based on that comparison, EF Core detects that a column has been added, and adds the appropriate migration.

You can now apply your migration as before:

.NET Core CLI

.NET CLI

```
dotnet ef database update
```

Note that this time, EF detects that the database already exists. In addition, when our first migration was applied above, this fact was recorded in a special migrations history table in your database; this allows EF to automatically apply only the new migration.

Excluding parts of your model

Sometimes you may want to reference types from another DbContext. This can lead to migration conflicts. To prevent this, exclude the type from the migrations of one of the DbContexts.

```C#
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<IdentityUser>()
        .ToTable("AspNetUsers", t => t.ExcludeFromMigrations());
}
```

Next steps

The above was only a brief introduction to migrations. Please consult the other documentation pages to learn more about managing migrations, applying them, and other aspects. The .NET Core CLI tool reference also contains useful information on the different commands

Additional resources

- Entity Framework Core tools reference - .NET Core CLI : Includes commands to update, drop, add, remove, and more.
- Entity Framework Core tools reference - Package Manager Console in Visual Studio : Includes commands to update, drop, add, remove, and more.
- .NET Data Community Standup session going over new migration features in EF Core 5.0.