Managing Migrations

Article • 09/12/2023

As your model changes, migrations are added and removed as part of normal development, and the migration files are checked into your project's source control. To manage migrations, you must first install the EF Core command-line tools.

> **Tip**
>
> If the `DbContext` is in a different assembly than the startup project, you can explicitly specify the target and startup projects in either the **Package Manager Console tools** or the **.NET Core CLI tools**.

Add a migration

After your model has been changed, you can add a migration for that change:

.NET Core CLI

.NET CLI

```
dotnet ef migrations add AddBlogCreatedTimestamp
```

The migration name can be used like a commit message in a version control system. For example, you might choose a name like *AddBlogCreatedTimestamp* if the change is a new `CreatedTimestamp` property on your `Blog` entity.

Three files are added to your project under the **Migrations** directory:

- **XXXXXXXXXXXXXX_AddCreatedTimestamp.cs**--The main migrations file. Contains the operations necessary to apply the migration (in `Up`) and to revert it (in `Down`).
- **XXXXXXXXXXXXXX_AddCreatedTimestamp.Designer.cs**--The migrations metadata file. Contains information used by EF.
- **MyContextModelSnapshot.cs**--A snapshot of your current model. Used to determine what changed when adding the next migration.

The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

Namespaces

You are free to move Migrations files and change their namespace manually. New migrations are created as siblings of the last migration. Alternatively, you can specify the directory at generation time as follows:

.NET Core CLI

.NET CLI

```
dotnet ef migrations add InitialCreate --output-dir Your/Directory
```

> **Note**
>
> You can also change the namespace independently of the directory using `--namespace`.

Customize migration code

While EF Core generally creates accurate migrations, you should always review the code and make sure it corresponds to the desired change; in some cases, it is even necessary to do so.

Column renames

One notable example where customizing migrations is required is when renaming a property. For example, if you rename a property from `Name` to `FullName`, EF Core will generate the following migration:

C#

```
migrationBuilder.DropColumn(
    name: "Name",
    table: "Customers");

migrationBuilder.AddColumn<string>(
    name: "FullName",
    table: "Customers",
    nullable: true);
```

EF Core is generally unable to know when the intention is to drop a column and create a new one (two separate changes), and when a column should be renamed. If the above migration is applied as-is, all your customer names will be lost. To rename a column, replace the above generated migration with the following:

C#

```
migrationBuilder.RenameColumn(
    name: "Name",
    table: "Customers",
    newName: "FullName");
```

Tip

The migration scaffolding process warns when an operation might result in data loss (like dropping a column). If you see that warning, be especially sure to review the migrations code for accuracy.

Adding raw SQL

While renaming a column can be achieved via a built-in API, in many cases that is not possible. For example, we may want to replace existing `FirstName` and `LastName` properties with a single, new `FullName` property. The migration generated by EF Core will be the following:

C#

```
migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");

migrationBuilder.AddColumn<string>(
    name: "FullName",
    table: "Customer",
    nullable: true);
```

As before, this would cause unwanted data loss. To transfer the data from the old columns, we rearrange the migrations and introduce a raw SQL operation as follows:

C#

```
migrationBuilder.AddColumn<string>(
    name: "FullName",
    table: "Customer",
    nullable: true);

migrationBuilder.Sql(
@"
    UPDATE Customer
    SET FullName = FirstName + ' ' + LastName;
");

migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");
```

Arbitrary changes via raw SQL

Raw SQL can also be used to manage database objects that EF Core isn't aware of. To do this, add a migration without making any model change; an empty migration will be generated, which you can then populate with raw SQL operations.

For example, the following migration creates a SQL Server stored procedure:

C#

```
migrationBuilder.Sql(
@"
    EXEC ('CREATE PROCEDURE getFullName
        @LastName nvarchar(50),
        @FirstName nvarchar(50)
    AS
        RETURN @LastName + @FirstName;')");
```

> **Tip**
>
> `EXEC` is used when a statement must be the first or only one in a SQL batch. It can also be used to work around parser errors in idempotent migration scripts that can occur when referenced columns don't currently exist on a table.

This can be used to manage any aspect of your database, including:

- Stored procedures

- Full-Text Search
- Functions
- Triggers
- Views

In most cases, EF Core will automatically wrap each migration in its own transaction when applying migrations. Unfortunately, some migrations operations cannot be performed within a transaction in some databases; for these cases, you may opt out of the transaction by passing `suppressTransaction: true` to `migrationBuilder.Sql`.

Remove a migration

Sometimes you add a migration and realize you need to make additional changes to your EF Core model before applying it. To remove the last migration, use this command.

.NET Core CLI

.NET CLI

```
dotnet ef migrations remove
```

After removing the migration, you can make the additional model changes and add it again.

> ⚠ **Warning**
>
> Avoid removing any migrations which have already been applied to production databases. Doing so means you won't be able to revert those migrations from the databases, and may break the assumptions made by subsequent migrations.

Listing migrations

You can list all existing migrations as follows:

.NET Core CLI

.NET CLI

```
dotnet ef migrations list
```

Checking for pending model changes

> **Note**
>
> This feature was added in EF Core 8.0.

Sometimes you may want to check if there have been any model changes made since the last migration. This can help you know when you or a teammate forgot to add a migration. One way to do that is using this command.

.NET CLI

```
dotnet ef migrations has-pending-model-changes
```

You can also perform this check programmatically using `context.Database.HasPendingModelChanges()`. This can be used to write a unit test that fails when you forget to add a migration.

Resetting all migrations

In some extreme cases, it may be necessary to remove all migrations and start over. This can be easily done by deleting your **Migrations** folder and dropping your database; at that point you can create a new initial migration, which will contain your entire current schema.

It's also possible to reset all migrations and create a single one without losing your data. This is sometimes called "squashing", and involves some manual work:

1. Back up your database, in case something goes wrong.
2. In your database, delete all rows from the migrations history table (e.g. `DELETE FROM [__EFMigrationsHistory]` on SQL Server).
3. Delete your **Migrations** folder.
4. Create a new migration and generate a SQL script for it (`dotnet ef migrations script`).
5. Insert a single row into the migrations history, to record that the first migration has already been applied, since your tables are already there. The insert SQL is the last

operation in the SQL script generated above, and resembles the following (don't forget to update the values):

```
SQL
```

```
INSERT INTO [__EFMigrationsHistory] ([MIGRATIONID], [PRODUCTVERSION])
VALUES (N'<full_migration_timestamp_and_name>', N'<EF_version>');
```

> ⚠ **Warning**
>
> Any custom migration code will be lost when the **Migrations** folder is deleted. Any customizations must be applied to the new initial migration manually in order to be preserved.

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.