Applying Migrations

Article • 02/18/2023

Once your migrations have been added, they need to be deployed and applied to your databases. There are various strategies for doing this, with some being more appropriate for production environments, and others for the development lifecycle.

> **Note**
>
> Whatever your deployment strategy, always inspect the generated migrations and test them before applying to a production database. A migration may drop a column when the intent was to rename it, or may fail for various reasons when applied to a database.

SQL scripts

The recommended way to deploy migrations to a production database is by generating SQL scripts. The advantages of this strategy include the following:

- SQL scripts can be reviewed for accuracy; this is important since applying schema changes to production databases is a potentially dangerous operation that could involve data loss.
- In some cases, the scripts can be tuned to fit the specific needs of a production database.
- SQL scripts can be used in conjunction with a deployment technology, and can even be generated as part of your CI process.
- SQL scripts can be provided to a DBA, and can be managed and archived separately.

.NET Core CLI

Basic Usage

The following generates a SQL script from a blank database to the latest migration:

.NET CLI

```
dotnet ef migrations script
```

With From (to implied)

The following generates a SQL script from the given migration to the latest migration.

.NET CLI

```
dotnet ef migrations script AddNewTables
```

With From and To

The following generates a SQL script from the specified `from` migration to the specified `to` migration.

.NET CLI

```
dotnet ef migrations script AddNewTables AddAuditTable
```

You can use a `from` that is newer than the `to` in order to generate a rollback script.

> ⚠ **Warning**
>
> Please take note of potential data loss scenarios.

Script generation accepts the following two arguments to indicate which range of migrations should be generated:

- The **from** migration should be the last migration applied to the database before running the script. If no migrations have been applied, specify `0` (this is the default).
- The **to** migration is the last migration that will be applied to the database after running the script. This defaults to the last migration in your project.

Idempotent SQL scripts

The SQL scripts generated above can only be applied to change your schema from one migration to another; it is your responsibility to apply the script appropriately, and only to databases in the correct migration state. EF Core also supports generating **idempotent** scripts, which internally check which migrations have already been applied (via the migrations history table), and only apply missing ones. This is useful if you don't

exactly know what the last migration applied to the database was, or if you are deploying to multiple databases that may each be at a different migration.

The following generates idempotent migrations:

.NET Core CLI

.NET CLI

```
dotnet ef migrations script --idempotent
```

Command-line tools

The EF command-line tools can be used to apply migrations to a database. While productive for local development and testing of migrations, this approach isn't ideal for managing production databases:

- The SQL commands are applied directly by the tool, without giving the developer a chance to inspect or modify them. This can be dangerous in a production environment.
- The .NET SDK and the EF tool must be installed on production servers and requires the project's source code.

> **Note**
>
> Each migration is applied in its own transaction. See GitHub issue #22616 for a discussion of possible future enhancements in this area.

.NET Core CLI

The following updates your database to the latest migration:

.NET CLI

```
dotnet ef database update
```

The following updates your database to a given migration:

.NET CLI

```
dotnet ef database update AddNewTables
```

Note that this can be used to roll back to an earlier migration as well.

> ⚠ **Warning**
>
> Please take note of potential data loss scenarios.

For more information on applying migrations via the command-line tools, see the EF Core tools reference.

Bundles

Migration bundles are single-file executables that can be used to apply migrations to a database. They address some of the shortcomings of the SQL script and command-line tools:

- Executing SQL scripts requires additional tools.
- The transaction handling and continue-on-error behavior of these tools are inconsistent and sometimes unexpected. This can leave your database in an undefined state if a failure occurs when applying migrations.
- Bundles can be generated as part of your CI process and easily executed later as part of your deployment process.
- Bundles can be executed without installing the .NET SDK or EF Tool (or even the .NET Runtime, when self-contained), and they don't require the project's source code.

.NET Core CLI

The following generates a bundle:

.NET CLI

```
dotnet ef migrations bundle
```

The following generates a self-contained bundle for Linux:

.NET CLI

```
dotnet ef migrations bundle --self-contained -r linux-x64
```

For more information on creating bundles see the EF Core tools reference.

`efbundle`

The resulting executable is named `efbundle` by default. It can be used to update the database to the latest migration. It's equivalent to running `dotnet ef database update` or `Update-Database`.

Arguments:

Expand table

Argument	Description
`<MIGRATION>`	The target migration. If '0', all migrations will be reverted. Defaults to the last migration.

Options:

Expand table

Option	Short	Description
`--connection <CONNECTION>`		The connection string to the database. Defaults to the one specified in AddDbContext or OnConfiguring.
`--verbose`	`-v`	Show verbose output.
`--no-color`		Don't colorize output.
`--prefix-output`		Prefix output with level.

The following example applies migrations to a local SQL Server instance using the specified username and password.

PowerShell

```
.\efbundle.exe --connection 'Data Source=(local)\MSSQLSERVER;Initial
Catalog=Blogging;User ID=myUsername;Password=myPassword'
```

⚠ Warning

Don't forget to copy appsettings.json alongside your bundle. The bundle relies on the presence of appsettings.json in the execution directory.

Migration bundle example

A bundle needs migrations to include. These are created using `dotnet ef migrations add` as described in *Create your first migration*. Once you have migrations ready to deploy, create a bundle using the `dotnet ef migrations bundle`. For example:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> dotnet ef migrations bundle
Build started...
Build succeeded.
Building bundle...
Done. Migrations Bundle: C:\local\AllTogetherNow\SixOh\efbundle.exe
PS C:\local\AllTogetherNow\SixOh>
```

The output is an executable suitable for your target operating system. In my case this is Windows x64, so I get an `efbundle.exe` dropped in my local folder. Running this executable applies the migrations contained within it:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> .\efbundle.exe
Applying migration '20210903083845_MyMigration'.
Done.
PS C:\local\AllTogetherNow\SixOh>
```

As with `dotnet ef database update` or `Update-Database`, migrations are applied to the database only if they have not been already applied. For example, running the same bundle again does nothing, since there are no new migrations to apply:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> .\efbundle.exe
No migrations were applied. The database is already up to date.
Done.
PS C:\local\AllTogetherNow\SixOh>
```

However, if changes are made to the model and more migrations are generated with `dotnet ef migrations add`, then these can be bundled into a new executable ready to apply. For example:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> dotnet ef migrations add SecondMigration
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS C:\local\AllTogetherNow\SixOh> dotnet ef migrations add Number3
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS C:\local\AllTogetherNow\SixOh> dotnet ef migrations bundle --force
Build started...
Build succeeded.
Building bundle...
Done. Migrations Bundle: C:\local\AllTogetherNow\SixOh\efbundle.exe
PS C:\local\AllTogetherNow\SixOh>
```

Tip

The `--force` option can be used to overwrite the existing bundle with a new one.

Executing this new bundle applies these two new migrations to the database:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> .\efbundle.exe
Applying migration '20210903084526_SecondMigration'.
Applying migration '20210903084538_Number3'.
Done.
PS C:\local\AllTogetherNow\SixOh>
```

By default, the bundle uses the database connection string from your application's configuration. However, a different database can be migrated by passing the connection string on the command line. For example:

.NET CLI

```
PS C:\local\AllTogetherNow\SixOh> .\efbundle.exe --connection "Data Source=
(LocalDb)\MSSQLLocalDB;Database=SixOhProduction"
Applying migration '20210903083845_MyMigration'.
Applying migration '20210903084526_SecondMigration'.
Applying migration '20210903084538_Number3'.
Done.
PS C:\local\AllTogetherNow\SixOh>
```

Note

This time, all three migrations were applied, since none of them had yet been applied to the production database.

Apply migrations at runtime

It's possible for the application itself to apply migrations programmatically, typically during startup. While productive for local development and testing of migrations, this approach is inappropriate for managing production databases, for the following reasons:

- If multiple instances of your application are running, both applications could attempt to apply the migration concurrently and fail (or worse, cause data corruption).
- Similarly, if an application is accessing the database while another application migrates it, this can cause severe issues.
- The application must have elevated access to modify the database schema. It's generally good practice to limit the application's database permissions in production.
- It's important to be able to roll back an applied migration in case of an issue. The other strategies provide this easily and out of the box.
- The SQL commands are applied directly by the program, without giving the developer a chance to inspect or modify them. This can be dangerous in a production environment.

To apply migrations programmatically, call `context.Database.Migrate()`. For example, a typical ASP.NET application can do the following:

C#

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var db =
scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
        db.Database.Migrate();
    }

    host.Run();
}
```

Note that `Migrate()` builds on top of the `IMigrator` service, which can be used for more advanced scenarios. Use `myDbContext.GetInfrastructure().GetService<IMigrator>()` to access it.

> ⚠ **Warning**
>
> - Carefully consider before using this approach in production. Experience has shown that the simplicity of this deployment strategy is outweighed by the issues it creates. Consider generating SQL scripts from migrations instead.
> - Don't call `EnsureCreated()` before `Migrate()`. `EnsureCreated()` bypasses Migrations to create the schema, which causes `Migrate()` to fail.