**Entity Framework Core**

## The model

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. For more information, see [Creating a Model](https://docs.microsoft.com/en-us/ef/core/modeling/).

EF supports the following model development approaches:

* Generate a model from an existing database.
* Hand code a model to match the database.
* Once a model is created, use [EF Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/) to create a database from the model. Migrations allow evolving the database as the model changes.

C#Copy

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace Intro

{

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(

@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted\_Connection=True");

}

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public int Rating { get; set; }

public List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public Blog Blog { get; set; }

}

}

## Querying

Instances of your entity classes are retrieved from the database using [Language Integrated Query (LINQ)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/). For more information, see [Querying Data](https://docs.microsoft.com/en-us/ef/core/querying/).

using (var db = new BloggingContext())

{

var blogs = db.Blogs

.Where(b => b.Rating > 3)

.OrderBy(b => b.Url)

.ToList();

}

## Saving data

Data is created, deleted, and modified in the database using instances of your entity classes. See [Saving Data](https://docs.microsoft.com/en-us/ef/core/saving/) to learn more.

C#Copy

using (var db = new BloggingContext())

{

var blog = new Blog { Url = "http://sample.com" };

db.Blogs.Add(blog);

db.SaveChanges();

}

## EF O/RM considerations

While EF Core is good at abstracting many programming details, there are some best practices applicable to any O/RM that help to avoid common pitfalls in production apps:

* Intermediate-level knowledge or higher of the underlying database server is essential to architect, debug, profile, and migrate data in high performance production apps. For example, knowledge of primary and foreign keys, constraints, indexes, normalization, DML and DDL statements, data types, profiling, etc.
* Functional and integration testing: It's important to replicate the production environment as closely as possible to:
  + Find issues in the app that only show up when using a specific versions or edition of the database server.
  + Catch breaking changes when upgrading EF Core and other dependencies. For example, adding or upgrading frameworks like ASP.NET Core, OData, or AutoMapper. These dependencies can affect EF Core in unexpected ways.
* Performance and stress testing with representative loads. The naïve usage of some features doesn't scale well. For example, multiple collections Includes, heavy use of lazy loading, conditional queries on non-indexed columns, massive updates and inserts with store-generated values, lack of concurrency handling, large models, inadequate cache policy.
* Security review: For example, handling of connection strings and other secrets, database permissions for non-deployment operation, input validation for raw SQL, encryption for sensitive data.
* Make sure logging and diagnostics are sufficient and usable. For example, appropriate logging configuration, query tags, and Application Insights.
* Error recovery. Prepare contingencies for common failure scenarios such as version rollback, fallback servers, scale-out and load balancing, DoS mitigation, and data backups.
* Application deployment and migration. Plan out how migrations are going to be applied during deployment; doing it at application start can suffer from concurrency issues and requires higher permissions than necessary for normal operation. Use staging to facilitate recovery from fatal errors during migration. For more information, see [Applying Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying).
* Detailed examination and testing of generated migrations. Migrations should be thoroughly tested before being applied to production data. The shape of the schema and the column types cannot be easily changed once the tables contain production data. For example, on SQL Server, nvarchar(max) and decimal(18, 2) are rarely the best types for columns mapped to string and decimal properties, but those are the defaults that EF uses because it doesn't have knowledge of your specific scenario.

## Create a new project

dotnet new console -o EFGetStarted

cd EFGetStarted

## Install Entity Framework Core

To install EF Core, you install the package for the EF Core database provider(s) you want to target. This tutorial uses SQLite because it runs on all platforms that .NET Core supports. For a list of available providers, see [Database Providers](https://docs.microsoft.com/en-us/ef/core/providers/).

dotnet add package Microsoft.EntityFrameworkCore.Sqlite

## Create the model

Define a context class and entity classes that make up the model.

In the project directory, create **Model.cs** with the following code

using System;

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace EFGetStarted

{

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

public string DbPath { get; }

public BloggingContext()

{

var folder = Environment.SpecialFolder.LocalApplicationData;

var path = Environment.GetFolderPath(folder);

DbPath = System.IO.Path.Join(path, "blogging.db");

}

// The following configures EF to create a Sqlite database file in the

// special "local" folder for your platform.

protected override void OnConfiguring(DbContextOptionsBuilder options)

=> options.UseSqlite($"Data Source={DbPath}");

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public List<Post> Posts { get; } = new();

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public Blog Blog { get; set; }

}

}

Tip: This application intentionally keeps things simple for clarity. [Connection strings](https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-strings) should not be stored in the code for production applications. You may also want to split each C# class into its own file.

## Create the database

The following steps use [migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/) to create a database.

* Run the following commands:

.NET CLICopy

dotnet tool install --global dotnet-ef

dotnet add package Microsoft.EntityFrameworkCore.Design

dotnet ef migrations add InitialCreate

dotnet ef database update

This installs [dotnet ef](https://docs.microsoft.com/en-us/ef/core/cli/dotnet) and the design package which is required to run the command on a project. The migrations command scaffolds a migration to create the initial set of tables for the model. The database update command creates the database and applies the new migration to it.

## Create, read, update & delete

* Open Program.cs and replace the contents with the following code:

using System;

using System.Linq;

namespace EFGetStarted

{

internal class Program

{

private static void Main()

{

using (var db = new BloggingContext())

{

// Note: This sample requires the database to be created before running.

Console.WriteLine($"Database path: {db.DbPath}.");

// Create

Console.WriteLine("Inserting a new blog");

db.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });

db.SaveChanges();

// Read

Console.WriteLine("Querying for a blog");

var blog = db.Blogs

.OrderBy(b => b.BlogId)

.First();

// Update

Console.WriteLine("Updating the blog and adding a post");

blog.Url = "https://devblogs.microsoft.com/dotnet";

blog.Posts.Add(

new Post { Title = "Hello World", Content = "I wrote an app using EF Core!" });

db.SaveChanges();

// Delete

Console.WriteLine("Delete the blog");

db.Remove(blog);

db.SaveChanges();

}

}

}

}

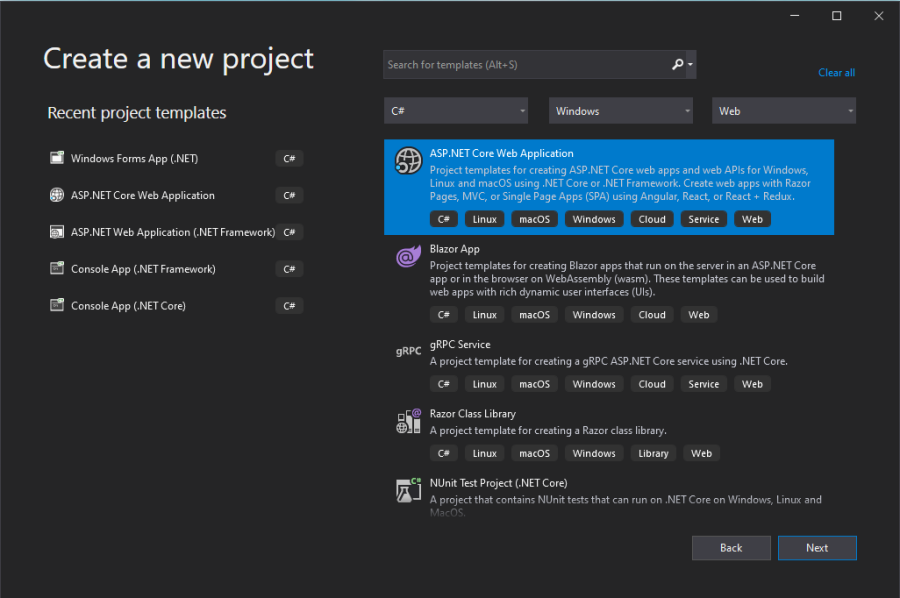
## Run the app

dotnet run

**ASP**.**NET Core** provides support to creating RESTful services, also known as **web APIs.**Entity Framework (EF) is an O/RM framework (Object/Relational Mapping). EF enables .NET developers to work with a database using .NET objects and provides an automated mechanism for accessing and storing the data in the database.

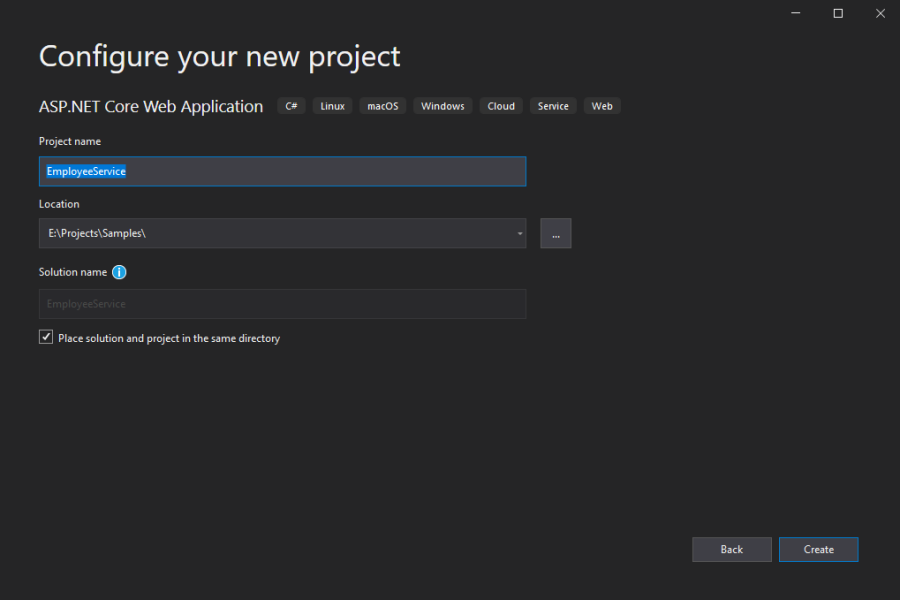
## Step 1: Creating a new Web API project

Open Visual Studio 2019 and click "Create a new project" and choose **ASP.NET Core Web Application.**



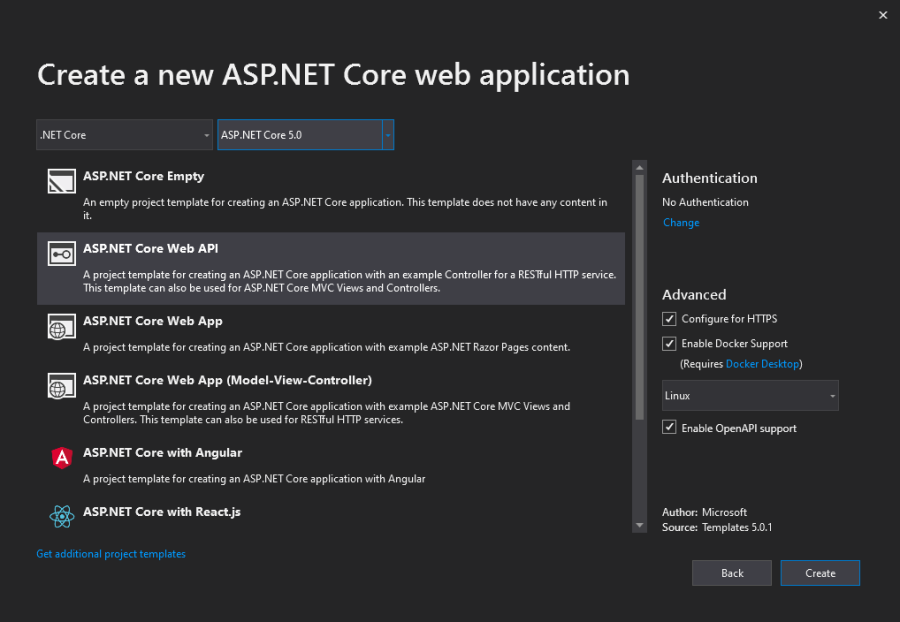
## Step 2:

In the next step, provide the project name as "EmployeeService" and select the location.

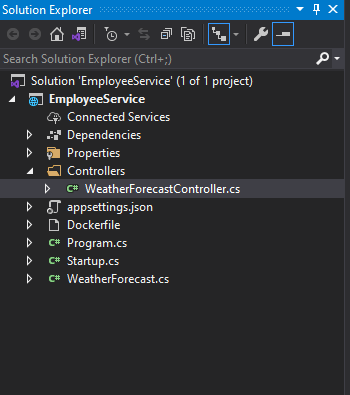


## Step 3:

Next, choose **ASP.NET Core 5.0** and ASP.NET Core Web API template and make sure that the “**Enable Docker Support**” option is selected with OS type as Linux. (We can also use IISExpress to run the Web API).

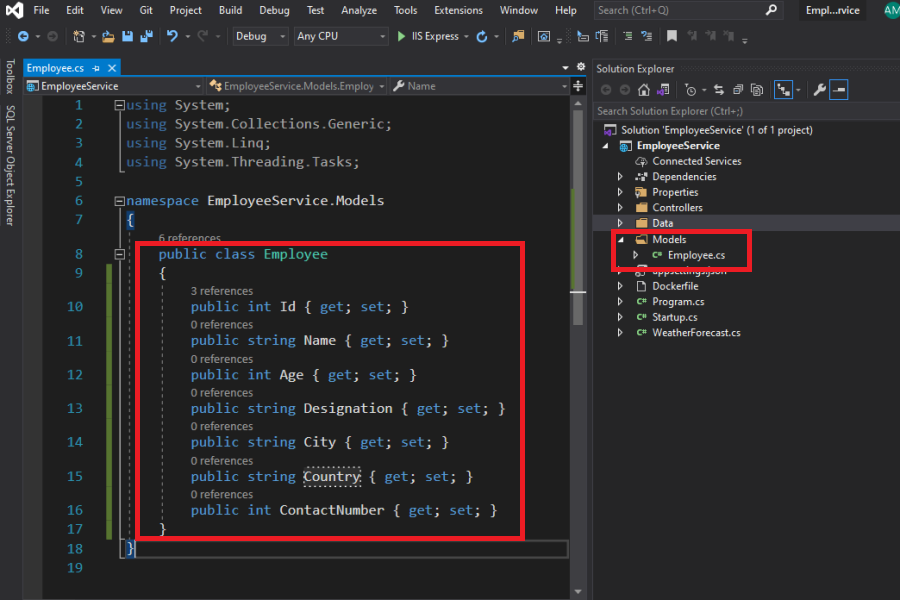


The solution will look like the below structure, by default "WeatherForecaseController" is created. We can remove this since it is not required and only for demo purposes.



## Step 4: Add Models

Add a new folder named “**Models**” to the project and add a class named "**Employee**" under the folder.



namespace EmployeeService.Models

{

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public string Designation { get; set; }

public string City { get; set; }

public string Country { get; set; }

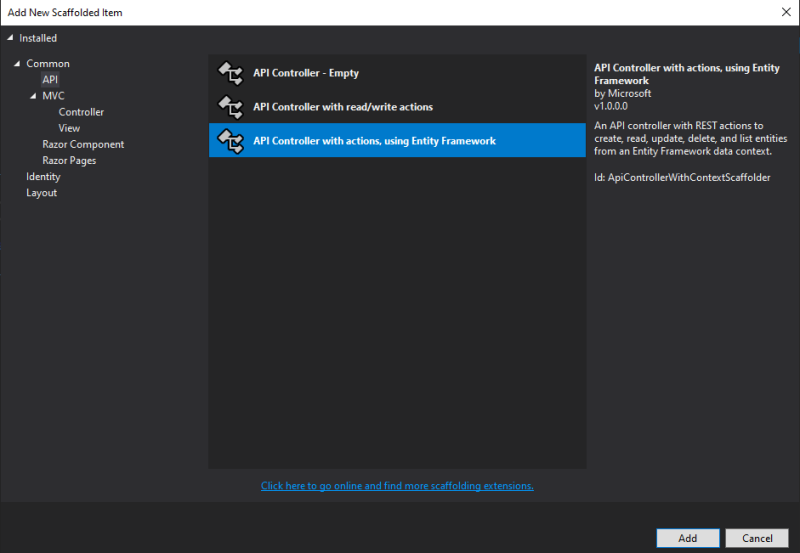
public int ContactNumber { get; set; }

}

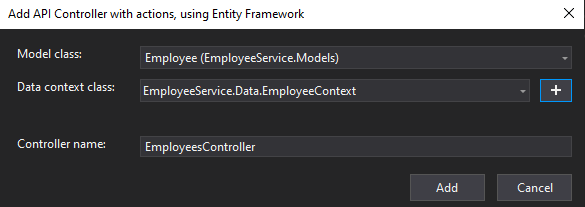
}

## Step 4: Add Entity Framework Database Context

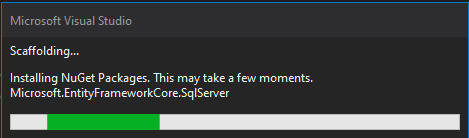
Right-click the Controllers folder and select Add -> Controller -> **API controller with actions, using Entity Framework**.



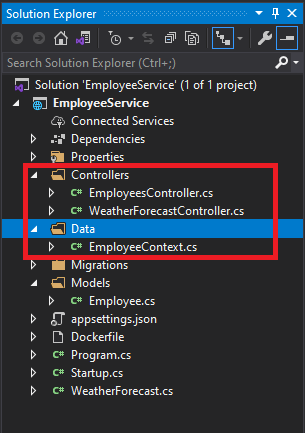
Select the model class name as **Employee**and Click the + button and add the **EmployeeContext.**



Once you click the **Add**button, Visual Studio will automatically add the necessary NuGet packages and create a new database context. (you will see the installation progress as shown below)



You will see the EmployeeContext file in the **Data**folder and **EmployeesController** with API actions under the Controller folder,



## EmployeesContext.cs

using Microsoft.EntityFrameworkCore;

namespace EmployeeService.Data

{

public class EmployeeContext : DbContext

{

public EmployeeContext (DbContextOptions<EmployeeContext> options)

: base(options)

{

}

public DbSet<EmployeeService.Models.Employee> Employee { get; set; }

}

}

## EmployeesController.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using EmployeeService.Data;

using EmployeeService.Models;

namespace EmployeeService.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class EmployeesController : ControllerBase

{

private readonly EmployeeContext \_context;

public EmployeesController(EmployeeContext context)

{

\_context = context;

}

// GET: api/Employees

[HttpGet]

public async Task<ActionResult<IEnumerable<Employee>>> GetEmployee()

{

return await \_context.Employee.ToListAsync();

}

// GET: api/Employees/5

[HttpGet("{id}")]

public async Task<ActionResult<Employee>> GetEmployee(int id)

{

var employee = await \_context.Employee.FindAsync(id);

if (employee == null)

{

return NotFound();

}

return employee;

}

// PUT: api/Employees/5

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPut("{id}")]

public async Task<IActionResult> PutEmployee(int id, Employee employee)

{

if (id != employee.Id)

{

return BadRequest();

}

\_context.Entry(employee).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!EmployeeExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// POST: api/Employees

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPost]

public async Task<ActionResult<Employee>> PostEmployee(Employee employee)

{

\_context.Employee.Add(employee);

await \_context.SaveChangesAsync();

return CreatedAtAction("GetEmployee", new { id = employee.Id }, employee);

}

// DELETE: api/Employees/5

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteEmployee(int id)

{

var employee = await \_context.Employee.FindAsync(id);

if (employee == null)

{

return NotFound();

}

\_context.Employee.Remove(employee);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool EmployeeExists(int id)

{

return \_context.Employee.Any(e => e.Id == id);

}

}

}

The above Employees controller, you will see the API methods like GET, POST, PUT and DELETE. This code will work without any modification. All the methods returns the data in JSON format.

In RESTful API, we will use the GetEmployee method to get the data from the database, PostEmployee to insert the data, PutEmployee to update the data, and DeleteEmployee to delete the data from the database.

## Step 5: Setup Migration

Before migration you have to modify the database connection, this will be found in appsettings.json file as shown below,

"ConnectionStrings": {

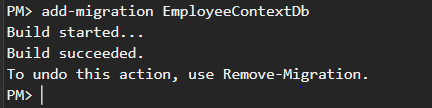
"EmployeeContext": "Server=localhost\\SQLExpress;Database=EmployeeContextDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

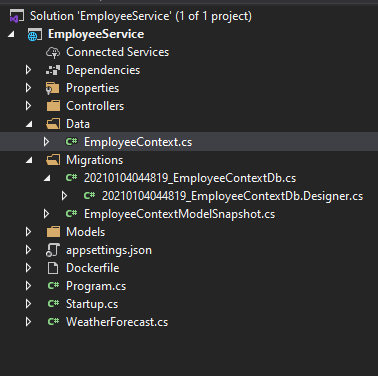
The above connection string was created when we create the Entity Framework Database Context. You can also use your localDb instead of a SQL database. But here in this example, we will use SQL database only.

To enable migrations, Click Tools -> NuGet Package Manager -> Package Manager Console,

Run this command, Add-Migration EmployeeContextDb



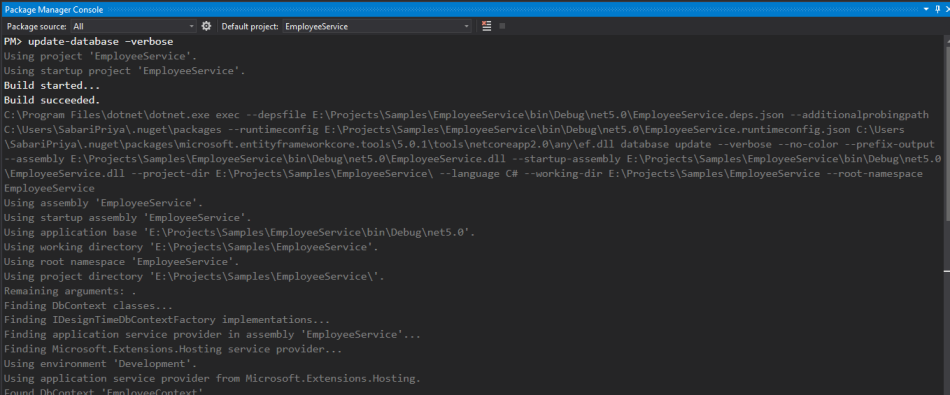
You will see the above message once the command ran successfully and also you will see the filer 20210104044819\_EmployeeContextDb.cs and EmployeeContextModelSnapshot.cs under the **Migrations**folder (see below).



## Step 6: Create a Database

You have to execute the below command in Package Manager Console to create a database,

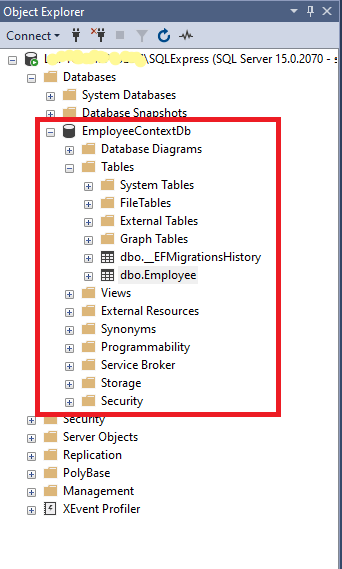
Update-Database -verbose



Once the above command is executed, the database and tables (EmployeeContextDb) will be created in an SQL server. (see below).

Open Microsoft SQL Server Management Studio -> Object Explorer -> Databases.

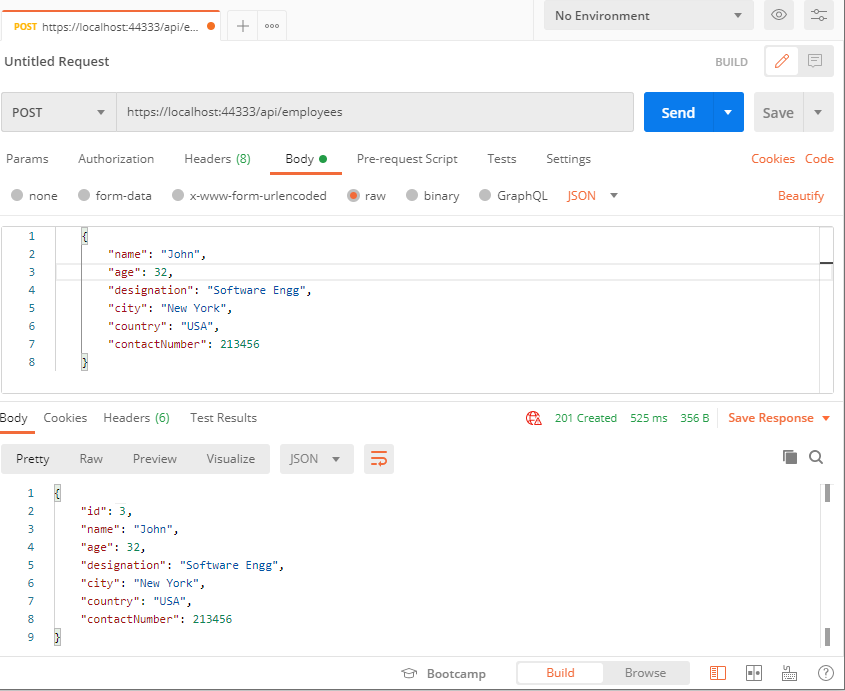
For any changes to the data model, you should use the Add-Migration MigrationName and Update-Database commands to push changes to the database.



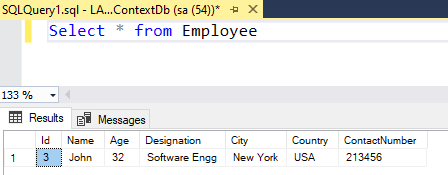
## Step 7: Test the API methods in Postman

**Add Employee (POST):**

Use POST method to add employee records to the database, enter this URL  **https://localhost:44333/api/employees** and select the method as **POST**and in the Body section, add a request JSON (as shown below) and click **Send**button.

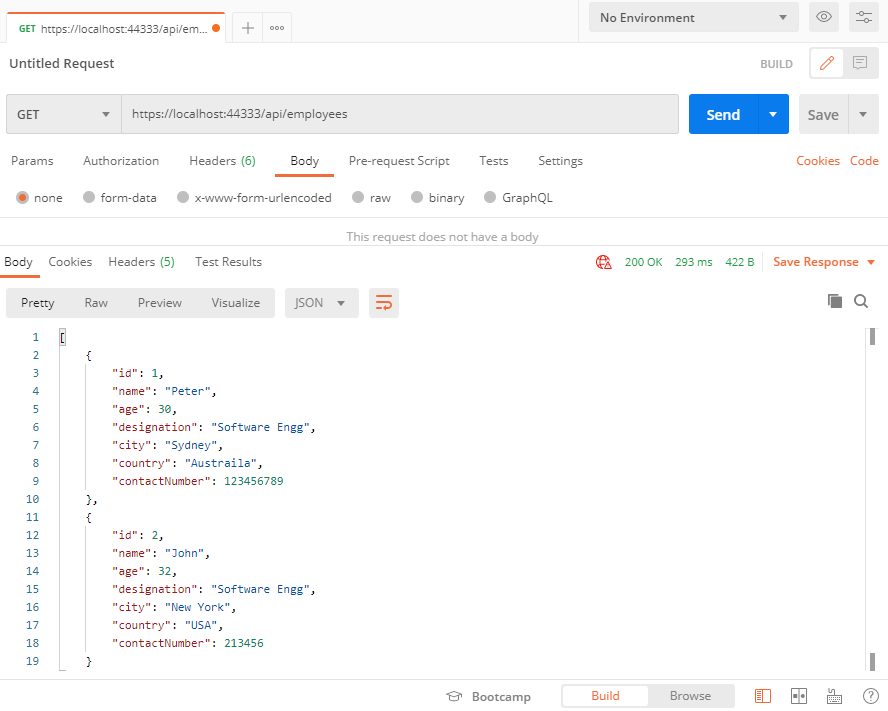


You can see the newly added employee record in SQL database (as shown below),



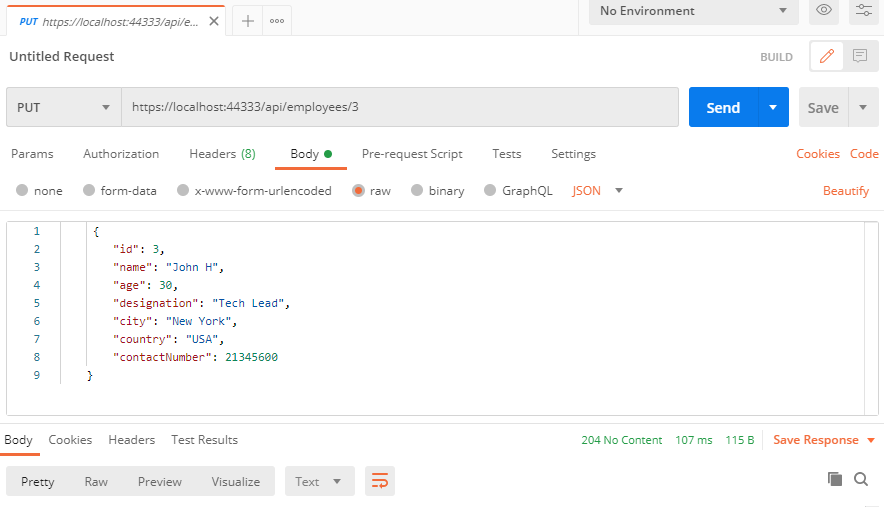
**Get Employees (GET):**

Use the GET method to retrieve all employees from the database, enter this URL  **https://localhost:44333/api/employees** and select the method as **GET**and click **Send**button.

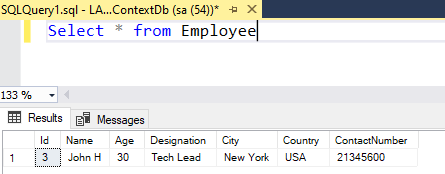


**Update Employee (PUT):**

Use PUT method to update employee records to the database, enter this URL  **https://localhost:44333/api/employees/{id}**(id -> should be employee id) and select the method as **PUT**and in the Body section, add a request JSON (as shown below) and click **Send**button.



You can see the updated employee details in SQL database (as shown below),



**Remove Employee (DELETE):**

Use DELETE method to delete an employee record from the database, enter this URL  **https://localhost:44333/api/employees/{id}**(id -> should be employee id) and select the method as **PUT**and in the Body section, add a request JSON (as shown below) and click **Send**button.

