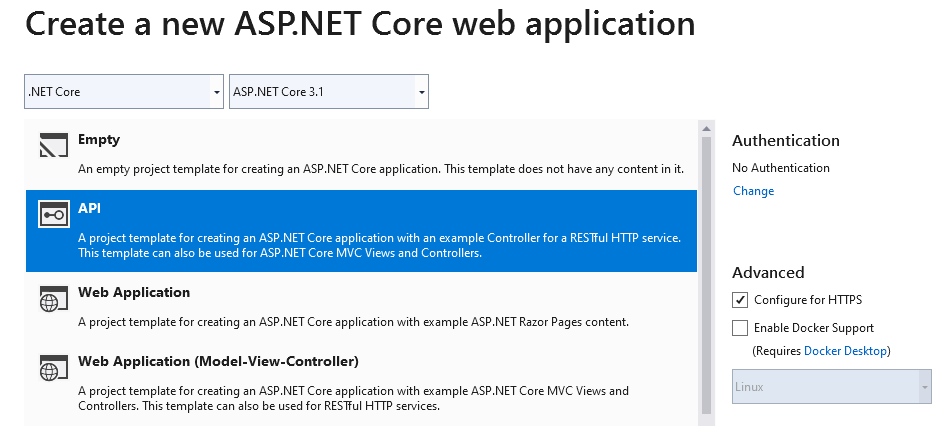
# Entity Framework Core in ASP.NET Core – Models, DbContext, Configuration

## Entities (Models) – The Classes That Map to the Database

Before we start talking about model classes, we need to create an ASP.NET Core project for the model classes to reside in. So, we are going to create an ASP.NET Core Web API project by using ASP.NET Core version 3.1:

**[](https://code-maze.com/wp-content/uploads/2020/01/Creating-ASP.NET-Core-App-EF-Core.png)**

We have to install Microsoft.EntityFrameworkCore and Microsoft.EntityFrameworkCore.SqlServer libraries. You can do that by using the NuGet Manager or with the Package Manager Console window.

The model (entity) class is a class that Entity Framework Core uses for mapping to the database table. So, let’s create our first entity and then explain how EF Core creates a table from the created class.

Let’s create a folder Entities and inside a class named Student:

public class Student

**{**

public Guid StudentId **{** get; set; **}**

public string Name **{** get; set; **}**

public int Age **{** get; set; **}**

**}**

There are a couple of rules that help EF Core to map this class to the database table and we are going to explain the whole process.

First of all, by using the name of our class, EF Core has the information which table to use for the columns and configuration mappings (as soon as we create the context class it will become much easier to understand how EF Core knows that).

Next, all the public properties from this class are going to be mapped to the table’s columns with the same names. Finally, EF Core uses a naming convention to create a primary key, from the StudentId property, in the database table (Later on, in the Configuring Nonerelational Properties section, we are going to see how).

## Providing the ConnectionString for EF Core in ASP.NET Core

Creating a model class or model classes is just one part of a puzzle. In order for EF Core to have the required information about the database to work with, we need to provide a connection string by modifying the appsettings.json file.

Placing a connection string into the appsettings.json file is a common practice (and we are going to use the same practice in this article) because we can use different appsettings files for different environments:

* Development.json – Holds the settings related to a development environment
* Production.json – Holds the settings related to a production environment
* appsettings.json –  Holds the settings that are common to both environments

But, we must state that using appsettings files to store connection string (or other sensitive data) isn’t the best practice, especially for the production environment. In such a case, the better way is to use environment variables.

Because we are not going to publish our app to the production environment, we are going to modify the appsettings.json file:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

**}**

**}**,

"ConnectionStrings": **{**

"sqlConnection": "server=.; database=CodeMaze; Integrated Security=true"

**}**,

"AllowedHosts": "\*"

**}**

As you can see, we are providing the database server name, the name of the database itself and the authorization information. Of course, you need to provide a valid name for the server part.

That is it, we can continue towards the context class.

## DbContext Class and its registration in ASP.NET Core

The context class is another important part of the application. This class must inherit from the DbContext base class which contains information and configuration for accessing the database. So, let’s create our context class named ApplicationContext in the Entities folder:

public class ApplicationContext : DbContext

**{**

public ApplicationContext**(**DbContextOptions options**)**

:base**(**options**)**

**{**

**}**

public DbSet**<**Student**>** Students **{** get; set; **}**

**}**

Additional options are sent to the base DbContext class through the ApplicationContext constructor by using DbContextOptions parameter. Finally, we see the Students property of type DbSet<Student> and this is quite an important part.

EF Core looks for all the public DbSet properties, inside the application’s context class, and then maps their names to the names of the tables in the database. Then it goes inside a class which is provided in the DbSet<T> property (in our case it is a Student class) and maps all the public properties to the table columns in the table with the same names and types (StudentId, Name, and Age).

If our Student class has any references towards other classes (right now it doesn’t but we will create relationships in the following articles), EF Core would use those reference properties and create relationships in the database.

### Registering a Context Class in the ASP.NET Core’s IOC

After we have finished with creating the ApplicationContext class, we can proceed towards its registration.

To do that, we are going to open the Startup.cs class and modify the ConfigureServices method:

public void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddDbContext**<**ApplicationContext**>(**opts =**>**

opts.UseSqlServer**(**Configuration.GetConnectionString**(**"sqlConnection"**)))**;

services.AddControllers**()**;

**}**

We are using the AddDbContext extension method to register our ApplicationContext class into the IOC container. Inside the UseSqlSrver method we are providing the connection string to our context class and we can provide additional options as well (when it comes to that).

### About AddDbContextPool

Instead of the AddDbContext method, we can use the AddDbContextPool method. We can use either the first or the second method, but with the second method, we enable DbContext pooling. This will not create a new instance every time but will check first if there are available instances in the pool and if there are, it will use one of those.

Right now our context class is ready to be used with the [**Dependency Injection (DI)**](https://code-maze.com/dependency-injection-aspnet/) inside our application. So let’s add the Values (API empty) controller and modify it:

public class ValuesController : ControllerBase

**{**

private readonly ApplicationContext \_context;

public ValuesController**(**ApplicationContext context**)**

**{**

\_context = context;

**}**

**[**HttpGet**]**

public IActionResult Get**()**

**{**

//Logic with the injected \_context object.

**}**

**}**

We can see that we are injecting our context class inside the controller’s constructor, which is the usual way for DI.

## Diving Deeper into the DbContext Class

Our ApplicationContext class currently accepts one parameter of type DbContextOptions inside a constructor. But we can provide the generic version of the DbContextOptions parameter as well:

public ApplicationContext**(**DbContextOptions**<**ApplicationContext**>** options**)**

:base**(**options**)**

**{**

**}**

Whether we use the generic or non-generic version of the DbContextOptions parameter, our application is going to work the same. The main difference is that the non-generic version is not recommended if our application has multiple context types, which is not the case right now.

If we navigate to the DbContext’s definition, we are going to see that it has three properties inside:

* **Database** – This property is responsible for the Transactions, Database Migrations/Creations and Raw SQL queries (we are going to talk about all of these in the following articles)
* **ChangeTracker** – This property is used to track states of the entities retrieved via the same context instance (this will be covered as well in the following articles)
* **Model** – This property provides access to the database model that EF Core uses when connecting or creating a database.

We can use the Model property to access the information for each entity and its properties.

But before we do that, let’s install the required Microsoft.EntityFrameworkCore.Relational package.

After the installation, we can modify our Get action:

var entity = \_context.Model

.FindEntityType**(**typeof**(**Student**)**.FullName**)**;

var tableName = entity.GetTableName**()**;

var schemaName = entity.GetSchema**()**;

var key = entity.FindPrimaryKey**()**;

var properties = entity.GetProperties**()**;

These are just a few examples of using the Model property, but still, we can see its purpose.

**ASP.NET Core Web API with EF Core Code-First Approach**

## Configuring EF Core

* [Defining the Model](https://code-maze.com/net-core-web-api-ef-core-code-first/#DefiningTheModel)
* [Creating a Context File](https://code-maze.com/net-core-web-api-ef-core-code-first/#CreatingContextFile)
* [Generating the Database from Code Using Migrations](https://code-maze.com/net-core-web-api-ef-core-code-first/#GeneratingDatabase)

### Defining the Model

First, let’s define the model. We will start by creating a folderModels within the root of the application.

Let’s add a new class Employee.cs inside:

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace EFCoreCodeFirstSample.Models

**{**

public class Employee

**{**

**[**Key**]**

**[**DatabaseGenerated**(**DatabaseGeneratedOption.Identity**)]**

public long EmployeeId **{** get; set; **}**

public string FirstName **{** get; set; **}**

public string LastName **{** get; set; **}**

public DateTime DateOfBirth **{** get; set; **}**

public string PhoneNumber **{** get; set; **}**

public string Email **{** get; set; **}**

**}**

**}**

The code above defines the classEmployee with some properties.

Additionally, we have decorated the EmployeeId property with Key and DatabaseGenerated attributes. We did this because we will be converting this class into a database table and the columnEmployeeId will serve as our primary key with the auto-incremented identity.

### Creating a Context File

As the next step, let’s create a context class, define database connection and register the context .let’s define the context file EmployeeContext.cs(it requires installed Microsoft.EntityFrameworkCore 3.0.0 package):

using Microsoft.EntityFrameworkCore;

namespace EFCoreCodeFirstSample.Models

**{**

public class EmployeeContext : DbContext

**{**

public EmployeeContext**(**DbContextOptions options**)**

: base**(**options**)**

**{**

**}**

public DbSet**<**Employee**>** Employees **{** get; set; **}**

**}**

**}**

and let’s define the database connection in the appsettings.json file as:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

**}**

**}**,

"ConnectionString": **{**

"EmployeeDB": "server=MY\_SERVER;database=EmployeeDB;User ID=MY\_USER;password=MY\_PASSWORD;"

**}**,

"AllowedHosts": "\*"

**}**

Of course, modify the ConnectionStringproperty to match with that of ours.

Then let’s install the Microsoft.EntityFrameworkCore.SqlServer package and register our context in the Startup.cs:

public void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddDbContext**<**EmployeeContext**>(**opts =**>** opts.UseSqlServer**(**Configuration**[**"ConnectionString:EmployeeDB"**]))**;

services.AddDbContext<EmployeeContext>(opt =>

opt.UseSqlServer(Configuration.GetConnectionString("MyConnection")));

services.AddControllers**()**;

**}**

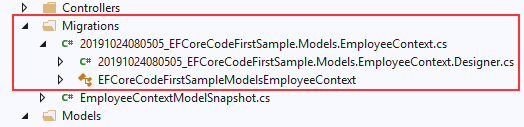
### Generating the Database from Code Using Migrations

Our next step is to add Code-First Migrations. Migrations automate the creation of database based on our Model. The EF Core packages required for migration will be added with .NET Core project setup.

Let’s install the Microsoft.EntityFrameworkCore.Tools package and run the following command in the Package Manager console:

PM> Add-Migration EFCoreCodeFirstSample.Models.EmployeeContext

This will create the classes for supporting migrations.

**[](https://code-maze.com/wp-content/uploads/2018/10/ef-core-migration-add-migration.png)**

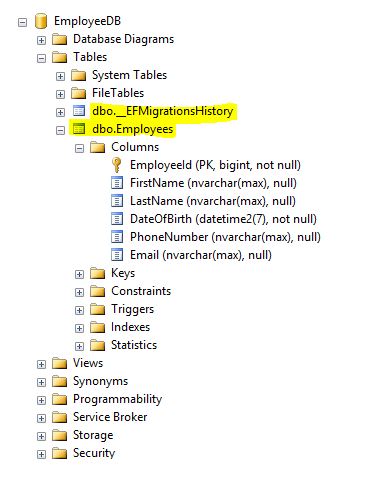
Now let’s apply those changes to the database.

Run the following command:

PM> update-database

This will update the database based on our models.

Now let’s verify that the database and tables are created by opening SQL Server Management Studio or Visual Studio Server Explorer:



We can see the database EmployeeDBis created with a table Employeeswhich contains the columns based on the fields we defined in our model.

Each time we make changes to our entities and do a migration, we can see new migration files created in our solution and new entries in the table\_\_EFMigrationsHistory.

When using the EF Core Code-First approach the best practice is to make all modifications to the database through the model and then update the database by doing the migration. Ideally, **we should not make any manual changes to the database.**

With that, the EF Core setup is complete.

## Seeding Data, Reverting Migrations and Creating DB Scripts

### Seeding Data

Data seeding allows us to provide initial data during the creation of a database. Then, EF Core migrations will automatically determine what insert, update or delete operations need to be applied when upgrading the database to a new version of the model.

So let’s create some seed data now. For this, we need to override the OnModelCreating method in the EmployeeContext class:

protected override void OnModelCreating**(**ModelBuilder modelBuilder**)**

**{**

modelBuilder.Entity**<**Employee**>()**.HasData**(**new Employee

**{**

EmployeeId = 1,

FirstName = "Uncle",

LastName = "Bob",

Email = "uncle.bob@gmail.com",

DateOfBirth = new DateTime**(**1979, 04, 25**)**,

PhoneNumber = "999-888-7777"

**}**, new Employee

**{**

EmployeeId = 2,

FirstName = "Jan",

LastName = "Kirsten",

Email = "jan.kirsten@gmail.com",

DateOfBirth = new DateTime**(**1981, 07, 13**)**,

PhoneNumber = "111-222-3333"

**})**;

**}**

Here we have provided two Employee records that will be inserted into the database as part of the migration.

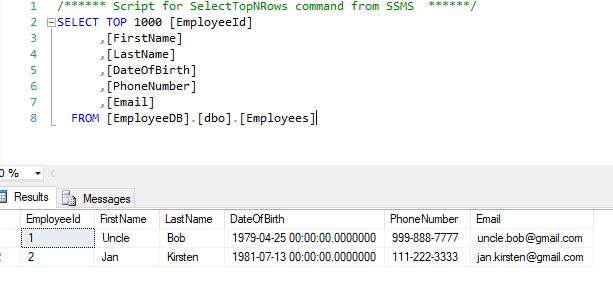
Let’s run the migration commands once again:

Add-Migration EFCoreCodeFirstSample.Models.EmployeeContextSeed

update-database

This will create a new migration file in our Migrations folder and update the database with the seed data we provided:

Now the Employee table in our database will look like this:



### Reverting Migrations

After making changes to our [**EF Core**](https://code-maze.com/entity-framework-core-series/) model, the database schema will be out of sync. To bring it to sync with the model, let’s add another migration.

Let’s add a new property Gender in our employee model and then do a migration.

It is a good practice to give meaningful names to the migration like a commit message in a version control system. For example, if we add a new field Gender to the Employee model, we may give a name like AddEmployeeGender.

Add-Migration EFCoreCodeFirstSample.Models.AddEmployeeGender

Sometimes we add a migration and then realize we need to make additional changes to our model before applying it. To remove the last migration, we can use the command:

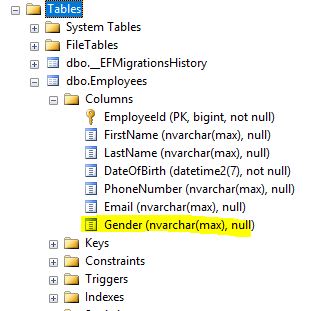
Remove-Migration

If we already applied a migration (or several migrations) to the database but need to revert it, we can use the same command to apply migrations, but specify the name of the migration we want to roll back to.

Let’s say we already applied the migration to add the Gender column to the database by using the below command.

update-database

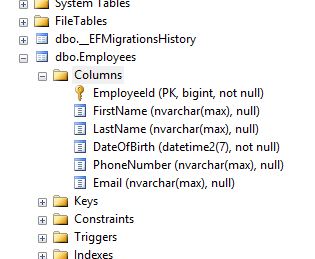
Now we can see the new column Gender added to the Employee table:



Now let’s say we want to revert this migration. We can use the same command by specifying the name of the previous migration:

update-database EFCoreCodeFirstSample.Models.EmployeeContextSeed

Once this is executed, we can see that the column Gender is removed from the Employee table:



We should remove the Gender property from the Employee class as well.

### Creating DB Scripts

While deploying our migrations to a production database, it’s useful to generate a SQL script. We can further tune the script to match the production database. Also, we can use the script along with various deployment tools.

The command to generate the script is:

Script-Migration

Once we apply this command, we can see a SQL script generated with all changes related to our migrations.

### Recommendation

If you want to learn in great detail about Entity Framework Core and many of its features, we recommend going through our [***Entity Framework Core series.***](https://code-maze.com/entity-framework-core-series/) Through the entire series, we talk about different EF Core features, from the Context classes and DbSet properties, relationships and none-relational configurations, additional migration information and querying the database. If you want, you have a place to learn a lot more about this topic.

## Creating the Repository

Now that we have configured the EF Core, we need a mechanism to access the data context from our API. Directly accessing the context methods from the API controller is a bad practice and we should avoid that.

So let’s implement a simple data repository using the repository pattern.

Let’s add a new folder under Models and name itRepository. Then let’s create a new interface calledIDataRepository:

namespace EFCoreCodeFirstSample.Models.Repository

**{**

public interface IDataRepository**<**TEntity**>**

**{**

IEnumerable**<**TEntity**>** GetAll**()**;

TEntity Get**(**long id**)**;

void Add**(**TEntity entity**)**;

void Update**(**TEntity dbEntity, TEntity entity**)**;

void Delete**(**TEntity entity**)**;

**}**

**}**

We will later inject this interface into our API Controller and API will be communicating with the data context using this interface.

Next, let’s create a concrete class that implements the interfaceIDataRepository. Let’s add a new folder under Models calledDataManager. Then let’s create a new class EmployeeManager:

using System.Collections.Generic;

using System.Linq;

using EFCoreCodeFirstSample.Models.Repository;

namespace EFCoreCodeFirstSample.Models.DataManager

**{**

public class EmployeeManager : IDataRepository**<**Employee**>**

**{**

readonly EmployeeContext \_employeeContext;

public EmployeeManager**(**EmployeeContext context**)**

**{**

\_employeeContext = context;

**}**

public IEnumerable**<**Employee**>** GetAll**()**

**{**

return \_employeeContext.Employees.ToList**()**;

**}**

public Employee Get**(**long id**)**

**{**

return \_employeeContext.Employees

.FirstOrDefault**(**e =**>** e.EmployeeId == id**)**;

**}**

public void Add**(**Employee entity**)**

**{**

\_employeeContext.Employees.Add**(**entity**)**;

\_employeeContext.SaveChanges**()**;

**}**

public void Update**(**Employee employee, Employee entity**)**

**{**

employee.FirstName = entity.FirstName;

employee.LastName = entity.LastName;

employee.Email = entity.Email;

employee.DateOfBirth = entity.DateOfBirth;

employee.PhoneNumber = entity.PhoneNumber;

\_employeeContext.SaveChanges**()**;

**}**

public void Delete**(**Employee employee**)**

**{**

\_employeeContext.Employees.Remove**(**employee**)**;

\_employeeContext.SaveChanges**()**;

**}**

**}**

**}**

The classEmployeeManager handles all database operations related to the employee. The purpose of this class is to separate the actual data operations logic from our API Controller.

This class has the following methods for supporting CRUD operations:

GetAll() – Gets all employee records from the database.

Get() – Gets a specific employee record from the database by passing an Id.

Add() – Creates a new employee record in the database.

Update() – Updates a specific employee record in the database.

Delete() – Removes a specific employee record from the database based on the Id.

As a next step, let’s configure the repository using dependency injection. This can be done in the ConfigureServices method in the Startup.cs as below:

public void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddDbContext**<**EmployeeContext**>(**opts =**>** opts.UseSqlServer**(**Configuration**[**"ConnectionString:EmployeeDB"**]))**;

services.AddScoped**<**IDataRepository**<**Employee**>**, EmployeeManager**>()**;

services.AddControllers**()**;

**}**

## Creating the API Controller

Now that our DataManager is all set, let’s create the API Controller and create the endpoints for handling CRUD operations. We have covered this in detail in one of our other articles: [Creating a .NET Core Web API Controller](https://code-maze.com/net-core-web-development-part5/#routing).

Following the above article, let’s create the EmployeeController class in the Controllersfolder as below:

using System.Collections.Generic;

using EFCoreCodeFirstSample.Models;

using EFCoreCodeFirstSample.Models.Repository;

using Microsoft.AspNetCore.Mvc;

namespace EFCoreCodeFirstSample.Controllers

**{**

**[**Route**(**"api/employee"**)]**

**[**ApiController**]**

public class EmployeeController : ControllerBase

**{**

private readonly IDataRepository**<**Employee**>** \_dataRepository;

public EmployeeController**(**IDataRepository**<**Employee**>** dataRepository**)**

**{**

\_dataRepository = dataRepository;

**}**

// GET: api/Employee

**[**HttpGet**]**

public IActionResult Get**()**

**{**

IEnumerable**<**Employee**>** employees = \_dataRepository.GetAll**()**;

return Ok**(**employees**)**;

**}**

// GET: api/Employee/5

**[**HttpGet**(**"{id}", Name = "Get"**)]**

public IActionResult Get**(**long id**)**

**{**

Employee employee = \_dataRepository.Get**(**id**)**;

if **(**employee == null**)**

**{**

return NotFound**(**"The Employee record couldn't be found."**)**;

**}**

return Ok**(**employee**)**;

**}**

// POST: api/Employee

**[**HttpPost**]**

public IActionResult Post**([**FromBody**]** Employee employee**)**

**{**

if **(**employee == null**)**

**{**

return BadRequest**(**"Employee is null."**)**;

**}**

\_dataRepository.Add**(**employee**)**;

return CreatedAtRoute**(**

"Get",

new **{** Id = employee.EmployeeId **}**,

employee**)**;

**}**

// PUT: api/Employee/5

**[**HttpPut**(**"{id}"**)]**

public IActionResult Put**(**long id, **[**FromBody**]** Employee employee**)**

**{**

if **(**employee == null**)**

**{**

return BadRequest**(**"Employee is null."**)**;

**}**

Employee employeeToUpdate = \_dataRepository.Get**(**id**)**;

if **(**employeeToUpdate == null**)**

**{**

return NotFound**(**"The Employee record couldn't be found."**)**;

**}**

\_dataRepository.Update**(**employeeToUpdate, employee**)**;

return NoContent**()**;

**}**

// DELETE: api/Employee/5

**[**HttpDelete**(**"{id}"**)]**

public IActionResult Delete**(**long id**)**

**{**

Employee employee = \_dataRepository.Get**(**id**)**;

if **(**employee == null**)**

**{**

return NotFound**(**"The Employee record couldn't be found."**)**;

**}**

\_dataRepository.Delete**(**employee**)**;

return NoContent**()**;

**}**

**}**

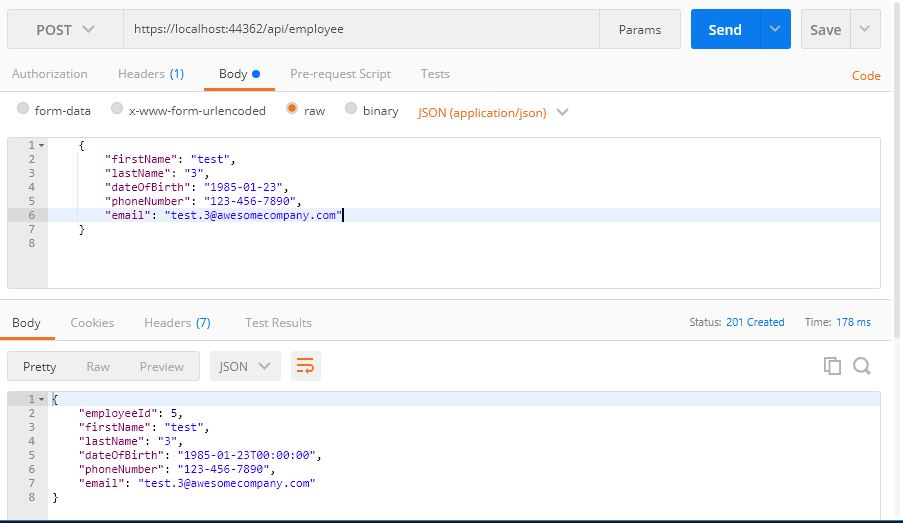
**}**

That’s it. We have successfully created a Web API controller with endpoints for handling CRUD operations.

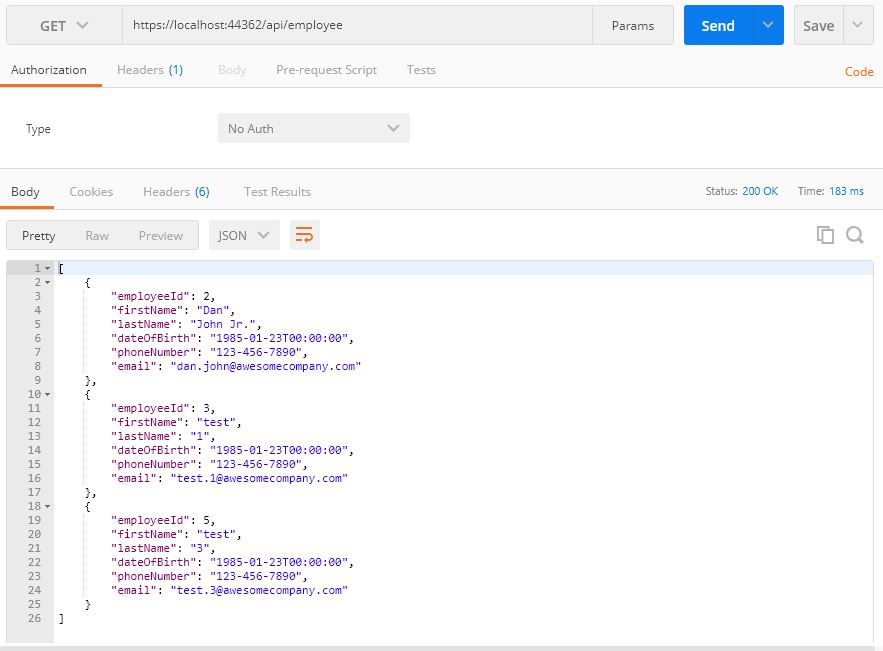
## Testing the API

Now let’s do a quick round of testing around our API endpoints using Postman.

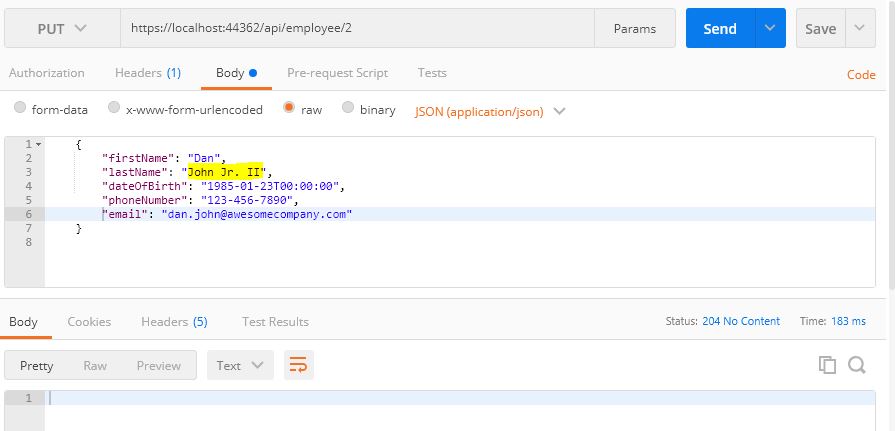
First, let’s create a new Employee using a Post request:



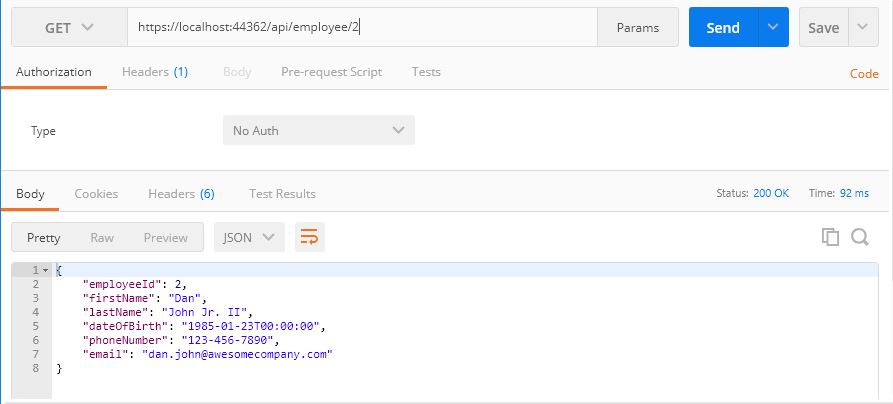
Next, let’s do a Get request to get all Employees. We can see the new Employee record which was created in the previous request:



Now, let’s do a Put request to test the update functionality by changing the last name:



Once again let’s do a Get request and verify that the last name has changed:



Now that we have successfully tested the API endpoints, let’s verify that the changes we made are actually persisted in the database. Let’s open the SQL Server management studio and verify that the record is created in the Employee table:

