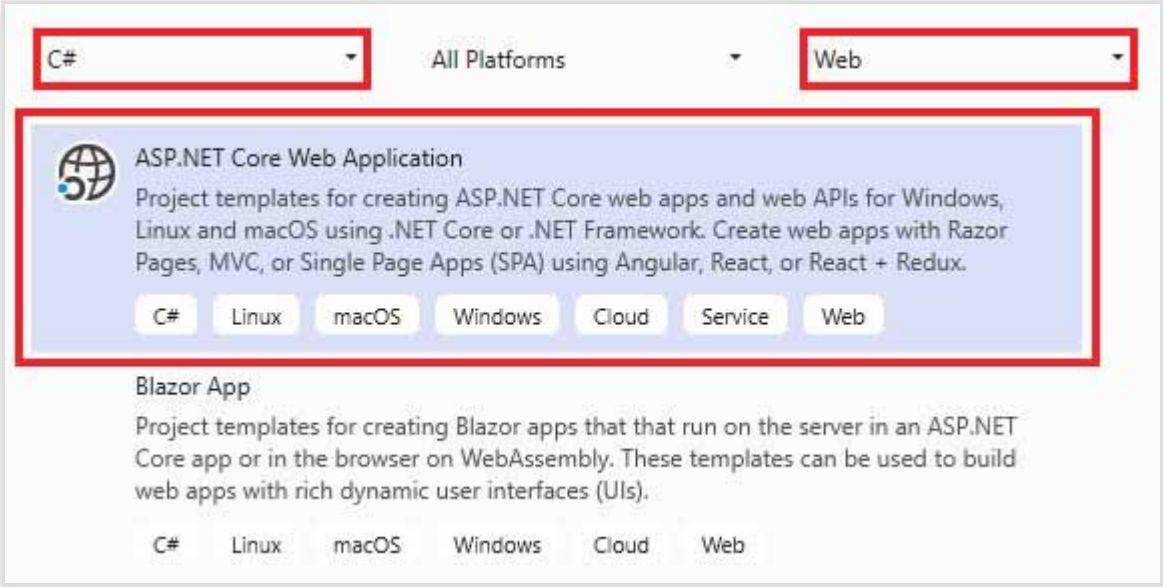
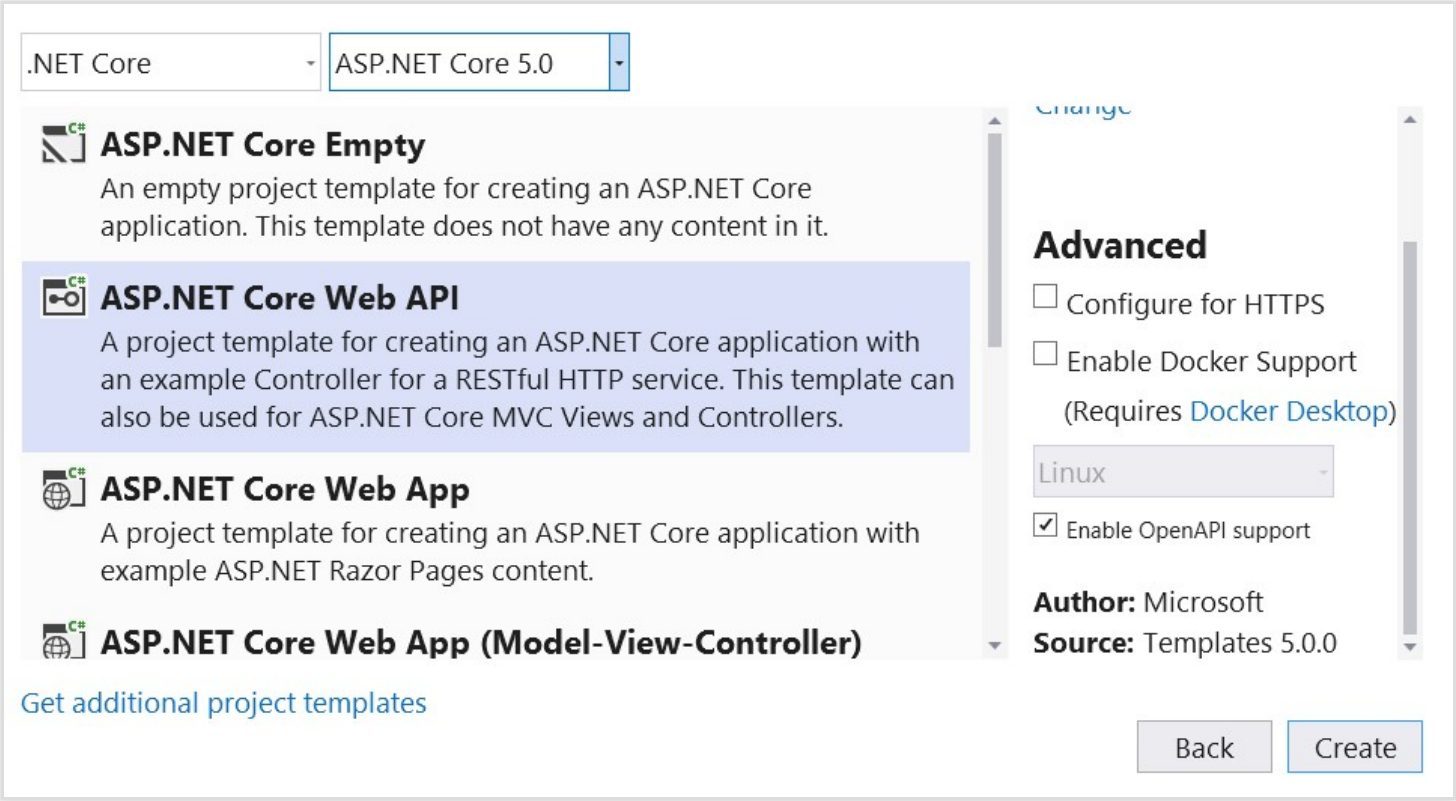
Create ASP.NET Core Web API

In Visual Studio 2019, From the new project window, select Asp.Net Core Web Application.

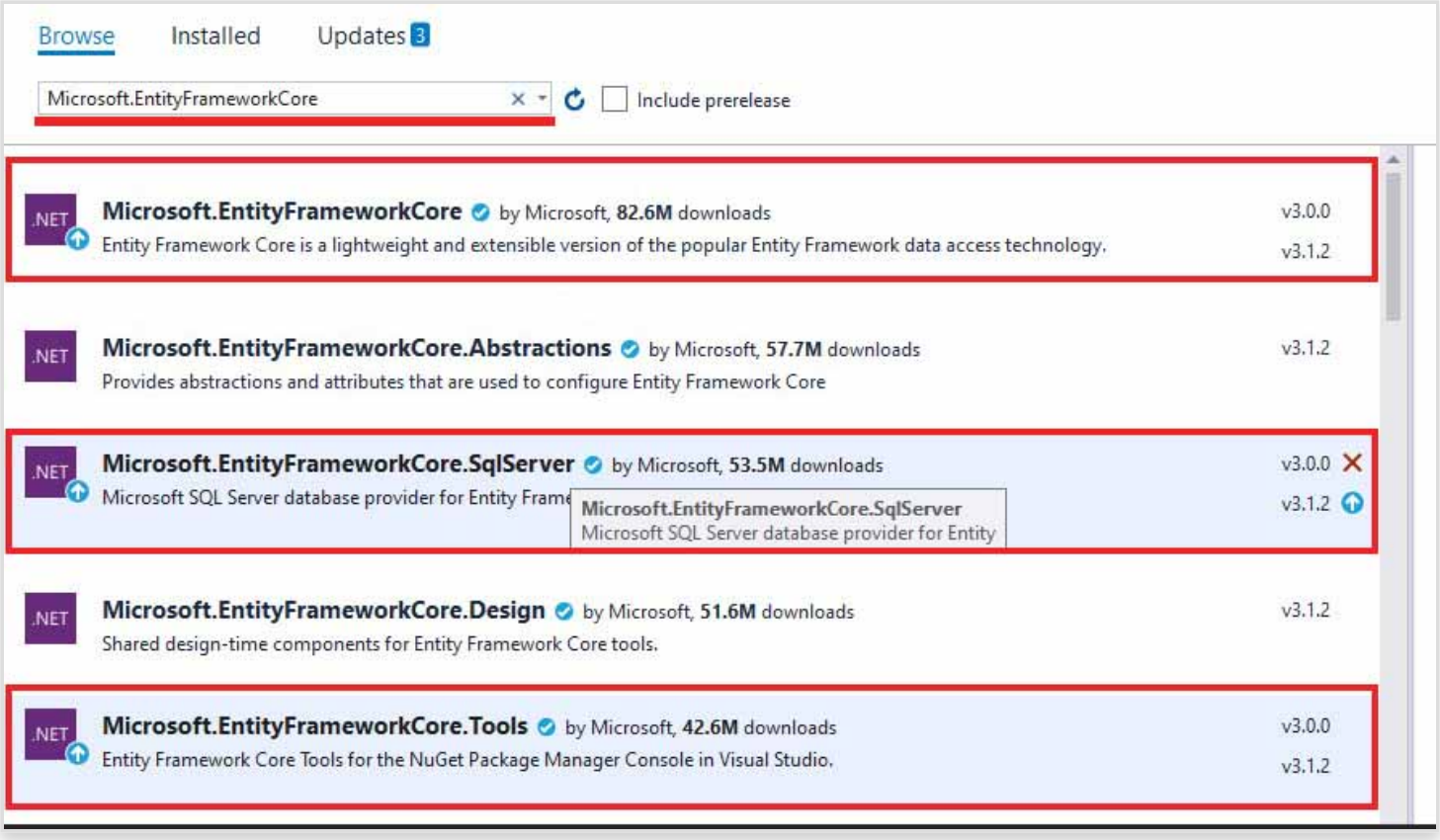


Once you provide the project name and location. A new window will be opened as follows, Select API. The above steps will create a brand new ASP.NET Core Web API project.



Setup Database

Let's create a Database for this project. Inside this project, we'll be using Entity Framework Core to create and interact with the database. So first of all we've to install corresponding NuGet packages. Right-click on the project name from Solution Explorer, click on Manage NuGet Packages, from Browse Tab, install the following 3 packages with same version as that of Asp.Net Core.



Now, let's define DB model class file – `PaymentDetail.cs` in a new folder `Models`.

C# Copy

```
public class PaymentDetail
{
    [Key]
    public int PaymentDetailId { get; set; }

    [Required]
    [Column(TypeName = "nvarchar(100)")]
    public string CardOwnerName { get; set; }

    [Required]
    [Column(TypeName = "varchar(16)")]
    public string CardNumber { get; set; }
```

```
    [Required]
    [Column(TypeName = "varchar(5)")]
    public string ExpirationDate { get; set; }

    [Required]
    [Column(TypeName = "varchar(3)")]
    public string SecurityCode { get; set; }
}
```

Now let's define `DbContext` class file- `/Models/PaymentDetailContext.cs`.

C# Copy

```
public class PaymentDetailContext : DbContext
{
    public PaymentDetailContext(DbContextOptions<PaymentDetailContext> opt
    { }

    public DbSet<PaymentDetail> PaymentDetails { get; set; }
}
```

DbContext class- `PaymentDetailContext` decides what should be added to actual physical database during DB Migration. So we have added `DbSet` property for `PaymentDetail` Model class, after migration `PaymentDetails` table will be created in SQL Server Database.

Into this model class constructor parameter- `options` , we have to pass which DbProvider (SQL Server, MySQL, PostgreSQL, etc) to use and corresponding DB connection string also. For that, we'll be using dependency injection in ASP.NET Core with `Startup.cs` file as follows.

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<PaymentDetailContext>(optionns =>
    optionns.UseSqlServer(Configuration.GetConnectionString("DevConnection"))
}
```

Here we've used dependency injection for `DbContext` class, through which SQL Server is set as a DbProvider with a connection string, Now save the connection string in `appsettings.json` file using `DevConnection` key as follows.

JSON Copy

```
{
  ....

  "ConnectionStrings": {
    "DevConnection": "Server=(local)\\sqlexpress;Database=PaymentDetailDB;Trusted_Connecti
  }
}
```

Now let's do the migration. Select project from solution explorer, then go to **Tools > NuGet Package Manager > Package Manager Console**. Then execute following commands one by one.

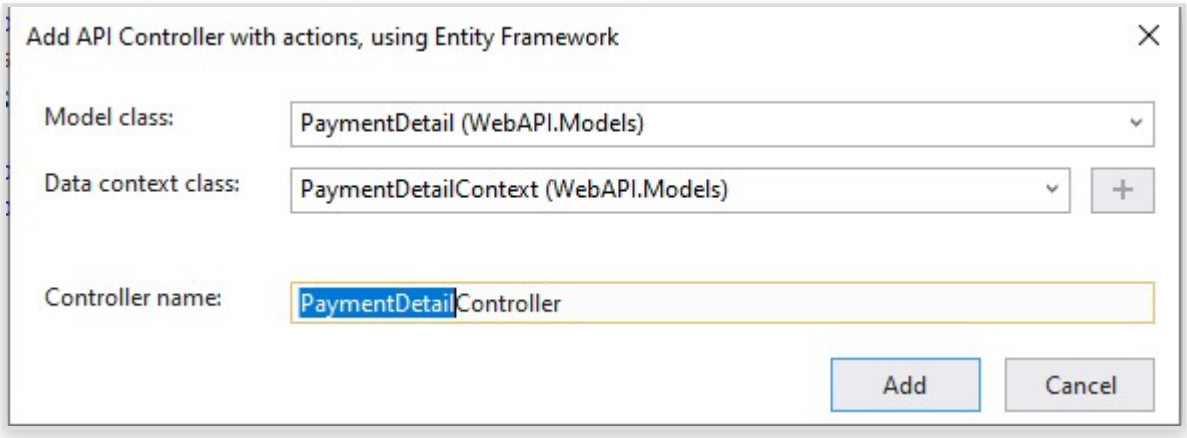
none Copy

```
Add-Migration "InitialCreate"
Update-Database
```

After successful migration, as per the connection string, a new database – `PaymentDetailDB` will be created with `PaymentDetails` table. Also, there will be a new `Migrations` folder created with corresponding C# files.

Create API Controller for CRUD Operations

To create a new API controller, right-click on Controllers folder **Add > Controller**, Select `API Controller with actions, using Entity Framework`.



With the help of scaffolding mechanism, newly created `PaymentDetailController` will look like this.

C# Copy

```
[Route("api/[controller]")]
[ApiController]
public class PaymentDetailController : ControllerBase
{
    private readonly PaymentDetailContext _context;

    public PaymentDetailController(PaymentDetailContext context)
    {
        _context = context;
    }

    // GET: api/PaymentDetail
    [HttpGet]
    public async Task<ActionResult<IEnumerable<PaymentDetail>>> GetPaymen
    { ... }

    // GET: api/PaymentDetail/5
    [HttpGet("{id}")]
    public async Task<ActionResult<PaymentDetail>> GetPaymentDetail(int id
    { ... }

    // PUT: api/PaymentDetail/5
    [HttpPut("{id}")]
    public async Task<IActionResult> PutPaymentDetail(int id, PaymentDetai
    { ... }

    // POST: api/PaymentDetail
    [HttpPost]
    public async Task<ActionResult<PaymentDetail>> PostPaymentDetail(Paym
    { ... }

    // DELETE: api/PaymentDetail/5
    [HttpDelete("{id}")]
    public async Task<ActionResult<PaymentDetail>> DeletePaymentDetail(in
```

```
    { ... }

    private bool PaymentDetailExists(int id)
    { ... }
}
```

It contains web methods POST, GET, PUT and DELETE for Create, Retrieve, Update and Delete operations respectively. As a constructor parameter we've `context` of the type `PaymentDetailContext` . the instance/value for this parameter will be passed from dependency injection from `StartUp` class.

For this project, we don't have to change anything in web methods and you can test any of these CRUD operations using software like postman or you use open api support with the swagger interface.