Technology Wonders

Using Model Binders in ASP.NET Core

December 09, 2019

In this post we will discuss Model Binders in ASP.NET Core applications. ASP.NET Core is a unified story for building Web UI using Razor Views and Web APIs. Web UI can be rendered using PageModel class (like we have Page class in WebForms) and Controllers (like MVC Controllers). The Razor pages, Controllers (MVC and Web API) works with data that comes from various HTTP requests. The data may be posted using HTTP Body, Route data values, Form Fields. When data is received from such variations then writing code to read such data and map this data with .NET CLR type on server-side is tedious and complex job. The **Model Binding** is used to simplify this complexity. Model Binding automates the process of mapping the data to .NET CLR types. The Model Binders provides following features

- It retrieves data received from Form fields, Http Body, Route parameters and query strings.
- The received data is provided to controllers as method parameters.
- The data is further converted in to .NET types and map with the complex object

In ASP.NET Core, we use following attribute classes to map the received data and map with CLR object

- FromBody, values are received from HTTP request body. This is useful to read data from HTTP POST and PUT request. The ASP.NET Core runtime delegates the responsibility of reading the HTTP request body to the i**nput formatter.** The Input Formatter is responsible to parse the data from the request body. In ASP.NET Core, the default format used by input formatter is JSON content type. We can certainly customize input formatters as per requirements (Its not in the scope of this post)
- FromRoute, values are received from the route data. This is good approach to read route values.
- FromQuery, values are received from the query string. Here I must mention that, its better to use FromRoute because the request URL is more simpler in case of route data.
- FromForm, values are received from the posted form field. This is a Key/Value pair for of posted data.
- FormHeader, values are received from Http Headers.

Using Model Binders in ASP.NET Core application

Step 1 : Open Visual Studio 2019 and create a ASP.NET Core application as shown in the following image

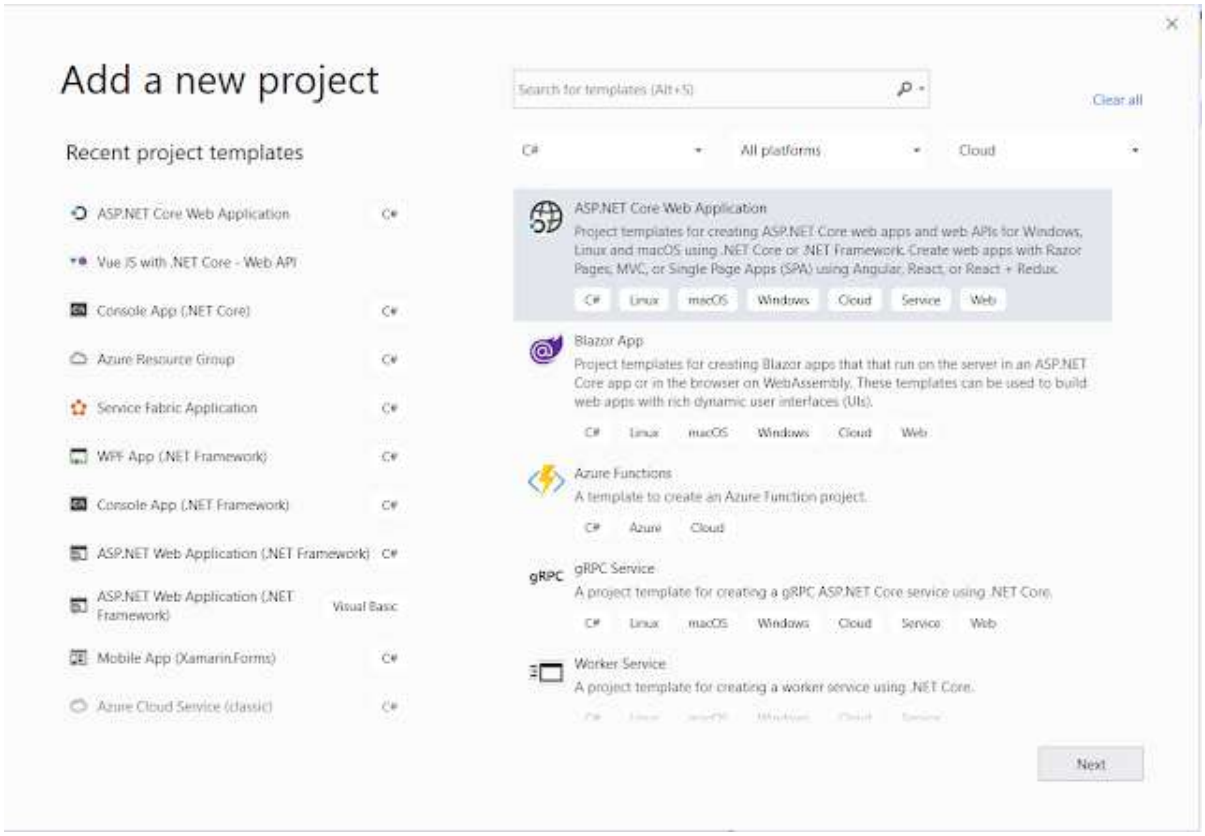


Image 1: Creating ASP.NET Core application

Name this application as **ParameterBinders.** Select the API Template for the project as shown in the following image

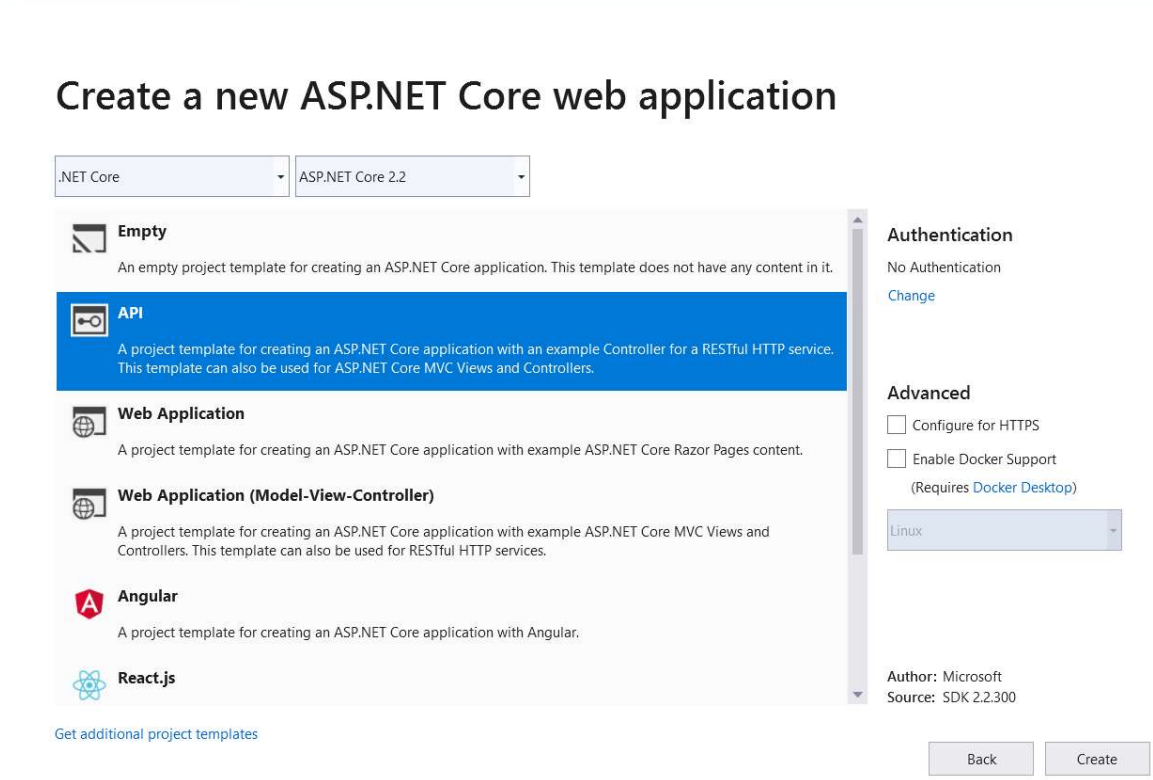


Image 2: The API Template

I am using API Template to demonstrate HTTP Post requests to read data from body using Model Binders.

Step 2: In the project add a new folder and name it as Models. In this folder add a new class file. Name this class file as Player.cs. Add the code in this file as shown in the following listing

```
using System.ComponentModel.DataAnnotations;

namespace ParameterBinders.Models
{
   public class Player
   {
      [Key]
      public int Id { get; set; }
      [Required(ErrorMessage ="Player Id is Must")]
      public string PlayerId { get; set; }
      [Required(ErrorMessage = "Player Name is Must")]
      public string PlayerName { get; set; }
      [Required(ErrorMessage = "Game is Must")]
      public string Game { get; set; }
   }
```

```
}
```

Listing 1: The Player class

We intent to store data in SQL Server database using Entity Framework Core (EF Core). We need to add DbContext class in the application (Note that, this post have used ASP.NET Core 2.2. This release by default provides packages like Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.SqlServer, Microsoft.EntityFrameworkCore.Relational, Microsoft.EntityFrameworkCore.Tools. If you are trying code from this post for ASP.NET Core 3.0+ then you need to install these packages explicitly.)

In the Models folder add a new class file and name it as PlayerContext.cs. In this class file add the code as shown in the following listing

```
using Microsoft.EntityFrameworkCore;

namespace ParameterBinders.Models
{
  public class PlayerContext : DbContext
  {
    public DbSet<Player> Players { get; set; }
    public PlayerContext(DbContextOptions<PlayerContext> options): base(options)
    {
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);
    }
  }
}
```

Listing 2: PlayerContext class

The above class declares the DbSet<Player>, this will map to the Players table after migration and database update using EF Core.

Step 3: Modify the appsettings.json to define database connection string as shown in the following listing

```
.....
"ConnectionStrings": {
```

```
  "AppConnStr": "Data Source=(localdb)\\mssqllocaldb;Initial Catalog=PlayerBinder;Integrated Security=SSPI"
 }
.....
```

(Note: Some code is removed from the file)

Listing 3: Connection string in appsettings.json

Step 4: Add a new folder in the project and name it as **Services.** In this folder add a new class file and name it as PlayerService.cs. In this service we will define a generic interface for declaring methods for create and read operations. The class PlayerService will implement this interface and will perform create and read operations using PlayerContext class. Add the code in this file as shown in the following listing

```
using Microsoft.EntityFrameworkCore;
using ParameterBinders.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ParameterBinders.Services
{

  public interface IService<T> where T: class
  {
    Task<IEnumerable<T>> GetAsync();
    Task<T> GetAsync(int id);
    Task<T> PostAsync(T data);
  }

  public class PlayerService : IService<Player>
  {
    private readonly PlayerContext context;
    public PlayerService(PlayerContext context)
    {
      this.context = context;
    }
    public async Task<IEnumerable<Player>> GetAsync()
    {
      return await context.Players.ToListAsync();
    }
```

```
        public async Task<Player> GetAsync(int id)
        {
            return await context.Players.FindAsync(id);
        }

        public async Task<Player> PostAsync(Player data)
        {
            try
            {
                var res =  await context.Players.AddAsync(data);
                await context.SaveChangesAsync();
                return res.Entity;

            }
            catch (Exception ex)
            {
                throw ex;
            }
        }
    }
}
```

Listing 4: The PlayerService class and its interface

All methods in the class are defined as Asynchronous methods. These methods uses PlayerContext class to perform Read and Write operations.

Step 5: We need to register PlayerContext and PlayerService classes in the Container registry using ConfigureServices() method of the Startup class as shown in the following listing

```
.....
services.AddDbContext<PlayerContext>
(options=>options.UseSqlServer(Configuration.GetConnectionString("AppConnStr")));
        services.AddScoped<IService<Player>, PlayerService>();
.....
```

(Note: Some code is removed.)

Listing 5: The ConfigureServices() method to register dependencies

Step 6: Now since we have already added DbContext and Service classes in the application and registered them in the dependency container, we can run the migrations. Open the

command prompt and run following commands to generate migrations and update database using the entity framework core (EFCore)

Command to generate migrations
dotnet ef migrations add playerMigration -c ParameterBinders.Models.PlayerContext
Command to update database
dotnet ef database update -c ParameterBinders.Models.PlayerContext

The **dotnet ef database update** command will generate database with **Players** table in it.

Step 7: Once database is generated, we can add controller in the application. Add a new empty MVC controller in Controllers folder. Name this controller as PlayerController. In this controller we will add various HttpPost methods. We will use model binders for the Post methods parameters. Add the code in this controller as shown in the following listing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ParameterBinders.Models;
using ParameterBinders.Services;
namespace ParameterBinders.Controllers
{
  [Route("api/[controller]/[action]")]
  [ApiController]
  public class PlayerController : ControllerBase
  {
    private readonly IService<Player> service;

    public PlayerController(IService<Player> service)
    {
      this.service = service;
    }

    [HttpGet]
    public IActionResult Get()
    {
      try
```

```
    {
        var res = service.GetAsync().Result;
        return Ok(res);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}

[HttpGet("{id}")]
public IActionResult Get(int id)
{
    try
    {
        var res = service.GetAsync(id).Result;
        return Ok(res);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}

[HttpPost]
[ActionName("PostByFromBody")]
public IActionResult PostByFromBody([FromBody] Player player)
{
    try
    {
        var res = service.PostAsync(player).Result;
        return Ok(res);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}

[HttpPost]
```

```
[ActionName("PostByParameters")]
public IActionResult PostByParameters(string playerId, string playerName, string game)
{
   try
   {
      var player = new Player()
      {
         PlayerId= playerId,
         PlayerName = playerName,
         Game = game
      };
      var res = service.PostAsync(player).Result;
      return Ok(res);
   }
   catch (Exception ex)
   {
      return BadRequest(ex.Message);
   }
}


[HttpPost]
[ActionName("PostByFromQuery")]
public IActionResult PostByFromQuery([FromQuery] Player player)
{
   try
   {
      var res = service.PostAsync(player).Result;
      return Ok(res);
   }
   catch (Exception ex)
   {
      return BadRequest(ex.Message);
   }
}

[HttpPost("{playerId}/{playerName}/{game}")]
[ActionName("PostByFromRoute")]
public IActionResult PostByFromRoute([FromRoute] Player player)
{
```

```
            try
            {
                var res = service.PostAsync(player).Result;
                return Ok(res);
            }
            catch (Exception ex)
            {
                return BadRequest(ex.Message);
            }
        }

        [HttpPost]
        [ActionName("PostByFromForm")]
        public IActionResult PostByFromForm([FromForm] Player player)
        {
            try
            {
                var res = service.PostAsync(player).Result;
                return Ok(res);
            }
            catch (Exception ex)
            {
                return BadRequest(ex.Message);
            }
        }
    }
}
```

Listing 6: The PlayerController

The **Route** attribute applied on the PlayerController class has the template as **api/[controller]/[action].** We will use this template to make call to the API with the method name e.g.
https://server/api/Player/PostByFromBody.

The above code have following HttpPost methods

1. PostByFormBody, this method accepts the Player model class as parameter. This parameter is decorated with **FromBody** attribute. This means that the posted data from HTTP request body will be mapped with Player, the CLR object.
2. PostByFromQuery, this method accepts the Player model class as parameter. This parameter is decorated with **FromQuery** attribute. This means that HTTP Posted data from query string will be mapped with the Player, the CLR object.

3. PostByFromRoute, this method accepts the Player model class as parameter. This parameter is decorated with **FromRoute** attribute. This means that HTTP Posted data from route parameters will be mapped with the Player input parameter.
4. PostbyFromForm, this method accepts the Player model class as parameter. This parameter is decorated with **FromForm** attribute. This means that the HTTP Post request from form fields will be mapped with the Player object.

To test all post methods apply breakpoint on each post method and run the application. We will use Postman tool to test these methods.

Open the Postman tool and make a new Post request as shown in the following image

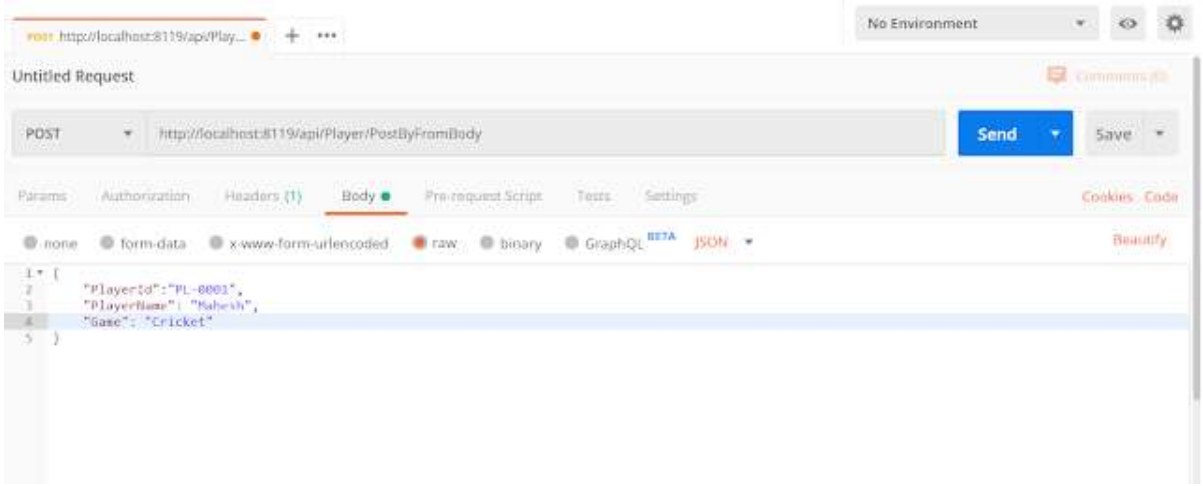


Image 3: The Post request for FromBody

Click on **Send** button, since we have already applied breakpoint on **PostByFromBody** method, we can see the data from HTTP request Body mapped with Player input parameter as shown in the following image

Image 4: The FromBody mapping

The data is mapped successfully with CLR object.

Make the new HTTP Post request for the **PostByFromQuery** method from Postman as shown in the following image

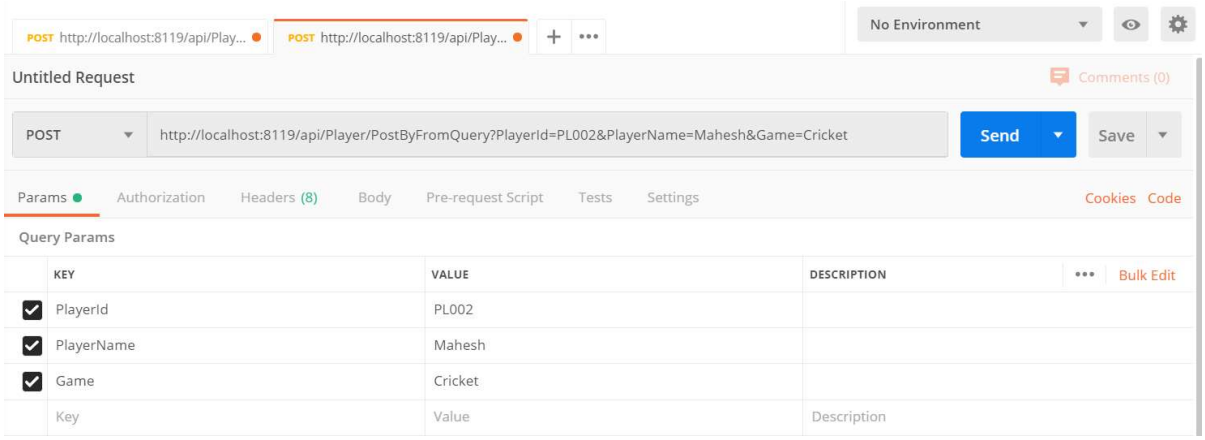


Image 5: The FromQuery request

Click on the **Send** button, we can see the mapping of the query string data to Player CLR object as shown in the following image

Image 6: The FromQuery mapping

Lets modify request in the Postman for **PostByFromRoute** method as shown in the following image

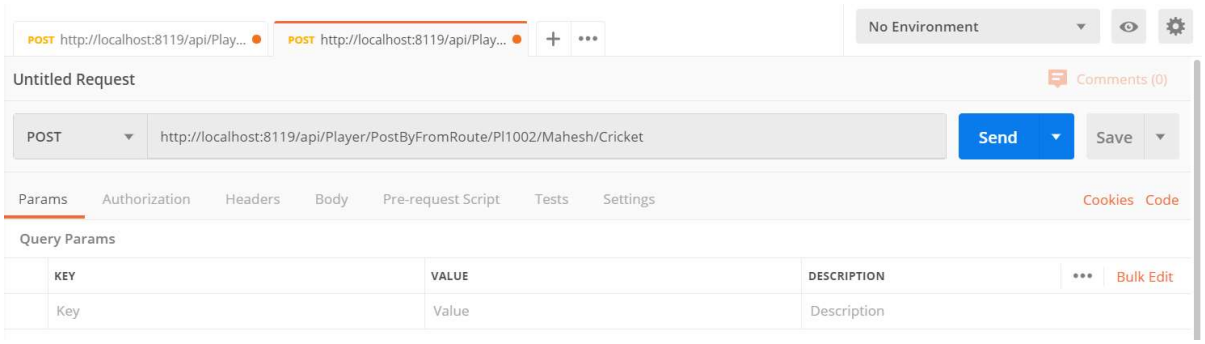


Image 7: The FormRoute request

Click on **Send** button, the data passed using route parameters will be mapped with Player object as shown in the following image

Image 8: The FormQuery mapping

The above image shows the mapping of the route parameters mapped with the CLR object. Likewise, we can test the **FromForm** mapping also.

We need not to write any code to map data from Route, Query String, etc. The mechanism is already provided by ASP.NET Core. The code become simple, The Model Binders is a great feature for implementing input formatters for HTTP posted data received from various client applications usign HTTP Body, Routes, etc.

That's it.

The code for this post can be downloaded from this link.

Conclusion: Using the Model binding, its easy for use to directly map the HTTP posted data to the .NET Types. This makes our code easier.

Popular posts from this blog

Configuring the React.js 16.8.7 application with WebPack 4.29.6 and Babel 7.4.3

August 04, 2019

Configuring the React.js 16.8.7 application with WebPack 4.29.6 and Babel 7.4.3

...

READ MORE

Understanding Token Based Authentication in ASP.NET Core 3.1 using JSON WEB TOKENS

January 03, 2020

In this post we will discuss the Token Based Authentication in ASP.NET Core 3.1. Security is one of the backbone for modern web application development. In most of the Modern Web Applications the sec ...

READ MORE

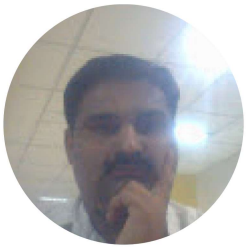
Using Session State in ASP.NET Core

December 08, 2019

In this post we will discuss the session management in ASP.NET Core applications. If you already have knowledge of ASP.NET Web Forms or ASP.NET MVC, then you must be knowing the Session state as ...

READ MORE

Powered by Blogger



MAHESH SABNIS

VISIT PROFILE

Archive

Report Abuse