.Net Core WebApi

1 Web Service

ASP.Net Core WebAPI ist für C# die Lösung zum Erstellen von WebServices. Derartige Server liefern nur Daten, kein HTML, Javascript oder CSS. Wenn man clientseitig mit jQuery oder Angular entwickelt, braucht man auf der Serverseite aber keine visuellen Komponenten mehr.
Hinweis: Bei C# gibt es auch die Variante ASP.Net Core MVC, die sowohl Daten als auch HTML/Javascript/Css bereitstellt. Das funktioniert mit der sogenannten Razor-Engine. Dies wird aber hier nicht besprochen.

Zu Beginn werden wir die Services im Backend sehr einfach programmieren, indem die Methoden „irgendwie“ benannt werden und einfach die entsprechenden Objekte zurückgegeben haben.
Nachteile:

- Etwaige Exceptions werden kaum oder gar nicht behandelt.
- Falls behandelt ➔die Information dazu wird als Property im Body mitgegeben und nicht im Header, der eigentlich dafür vorgesehen ist.
- Die Signaturen eines REST clients sind eigentlich genau definiert - daran sollte man sich halten.

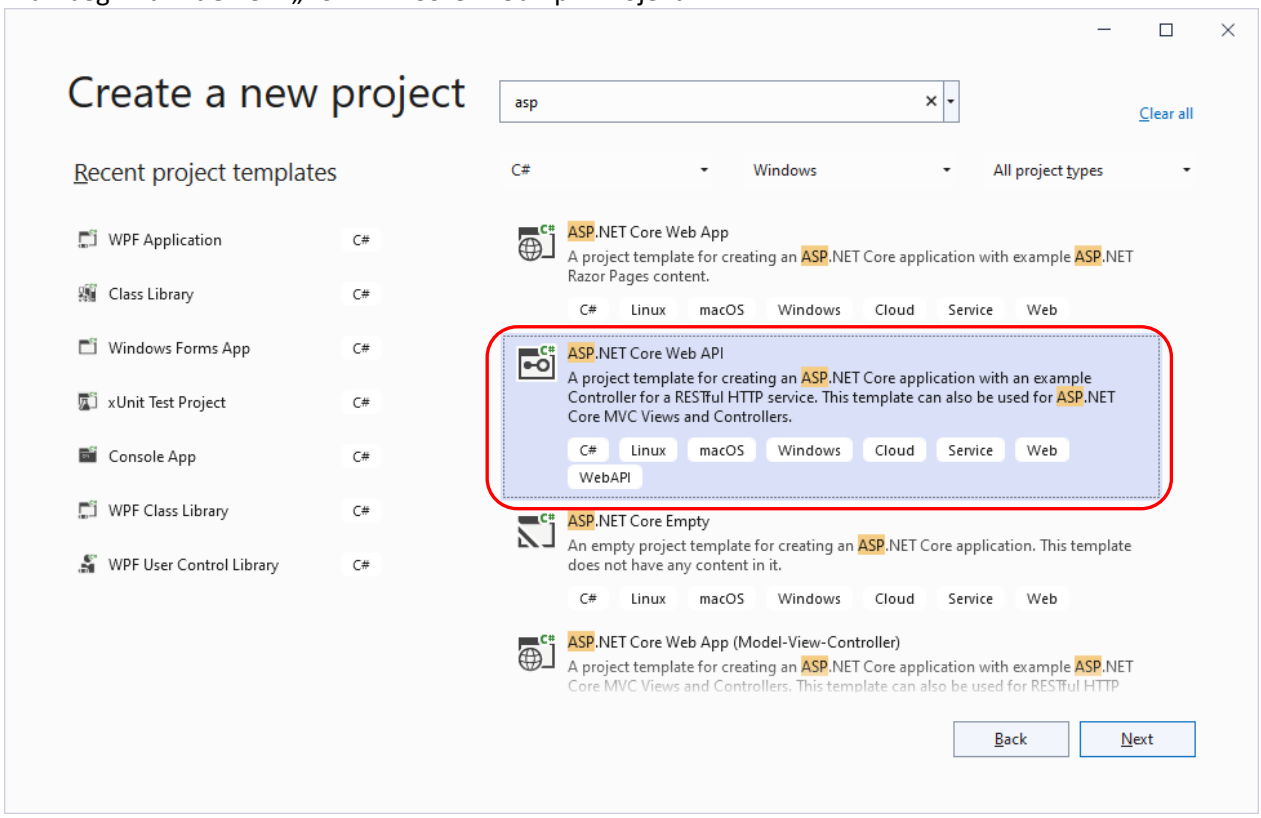
Diese Fehler werden in einem weiteren Tutorial behoben.

1.1 Testen

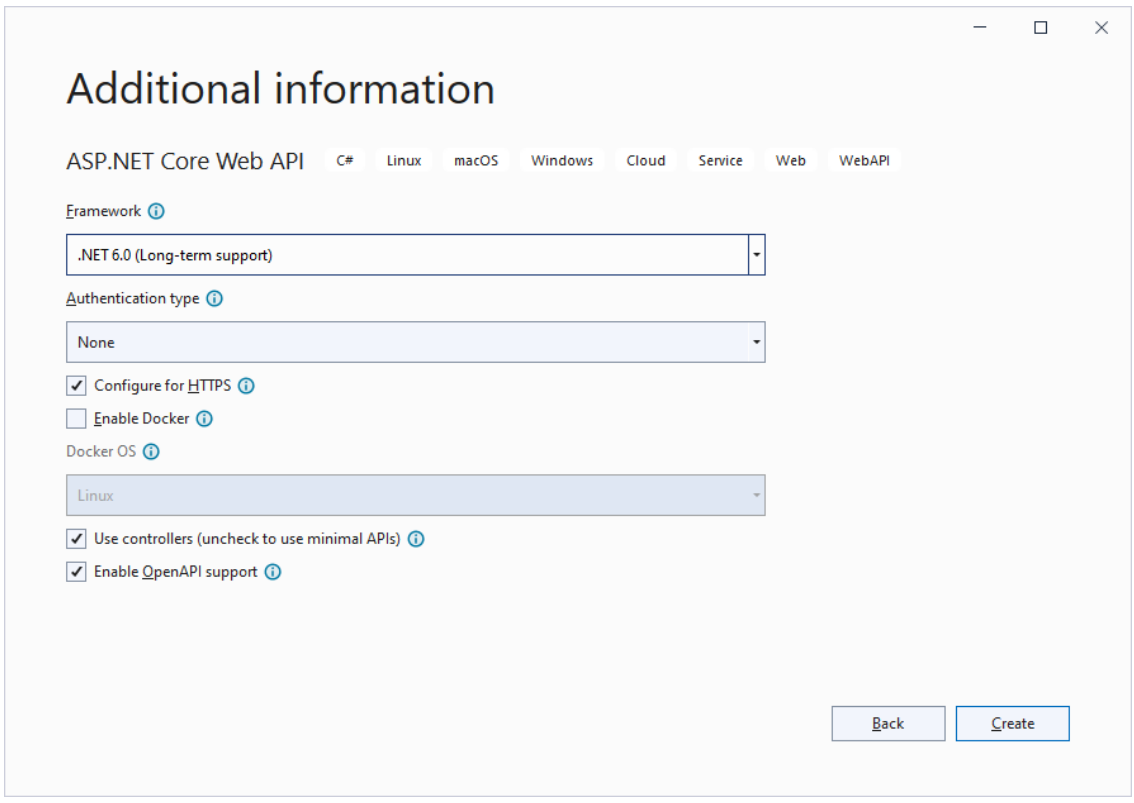
Hauptanwendung ist dabei der Zugriff auf derartige Webservices, hauptsächlich von Javascript aus. Die Signatur ergibt sich ja automatisch aus den Routen bzw. Parametern und Rückgabewerten der einzelnen Methoden.
Daher soll bald zu Beginn eine Möglichkeit gezeigt werden, wie man das relativ einfach testen kann und am Ende dann noch ein Tool, mit dem man die Schnittstellen automatisch erzeugen kann.

2 Visual Studio Projekt

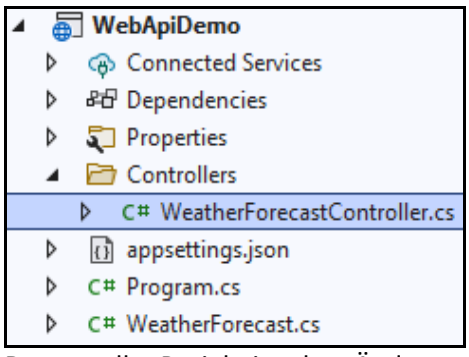
Vorerst soll ein ganz einfaches WebApi-Projekt erstellt und die einzelnen Komponenten besprochen werden. Man beginnt mit einem „ASP.NET Core Web Api“-Projekt:



Danach wie üblich das Projekt benennen (z.B. WebApiDemo) und Target auswählen



Es fällt auf, dass etwaige wwwroot-Folder komplett fehlen. Es sind wirklich nur noch die cs-Dateien vorhanden:



Das erstellte Projekt ist ohne Änderung sofort lauffähig.

2.1 WeatherController

Standardmäßig wird ein Controller namens WeatherForecastController.cs erstellt.

```
[ApiController]
[Route("[controller]")]
3 references
public class WeatherForecastController : ControllerBase
{
```

Hinweise:

- Die Basisklasse ist **`ControllerBase`**
- Die Klasse ist mit **`[ApiController]`** annotiert
- Der Name der Klasse endet üblicherweise auf „Controller“ (muss er aber nicht zwingend)
- Für die Klasse wird mit Attribute Routing eine Basis-Route festgelegt (siehe weiter unten).

2.2 Pakete

Etwaige Pakete sind wieder wie schon bei EF Core Database First beschrieben nach Bedarf auf eine der dort beschriebenen Weisen dem Projekt hinzuzufügen.

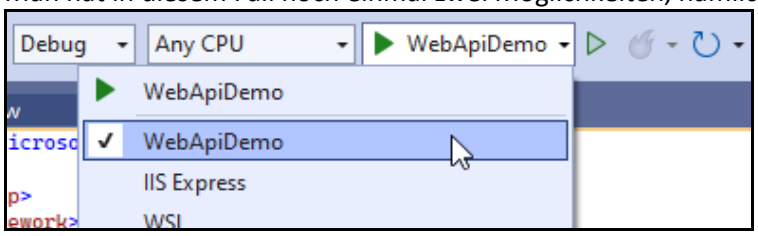
3 Starten

Hier gibt es zwei Möglichkeiten:

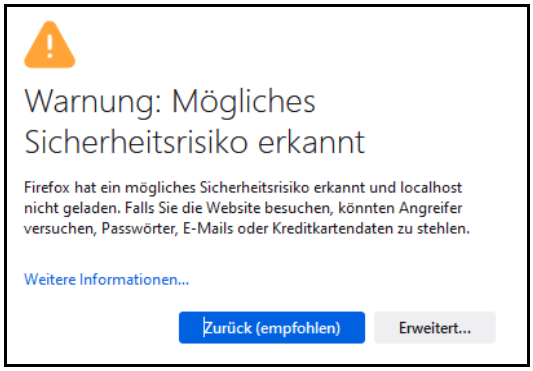
- Starten von der Konsole im Watch-Modus
- Starten im Visual Studio mit Debugger

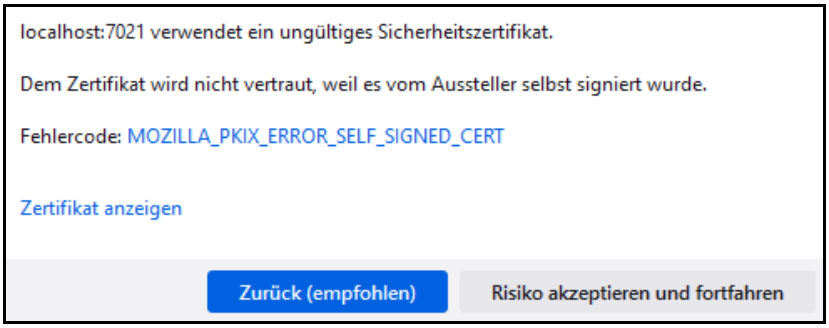
3.1 Start in Visual Studio

Man hat in diesem Fall noch einmal zwei Möglichkeiten, nämlich mit IIS Express oder als Konsolenanwendung:



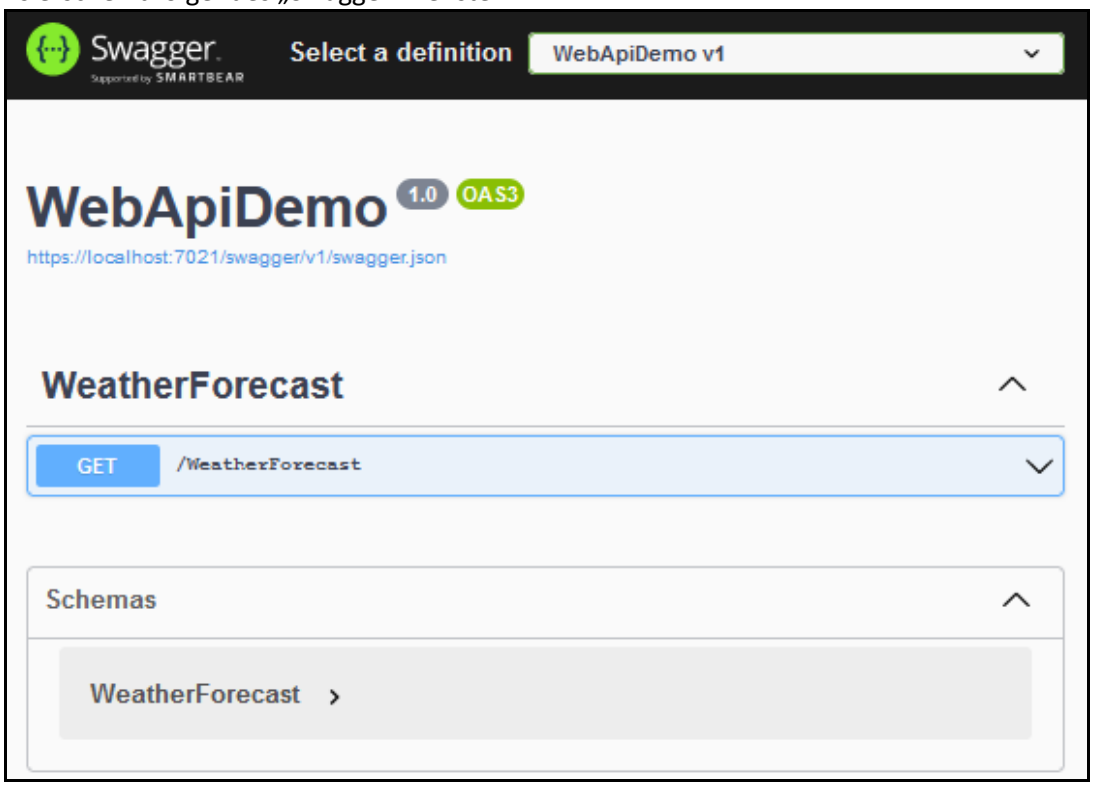
Davor erscheinen aufgrund von SSL noch folgende Dialoge, die man einfach abnicken kann.
Auf „Erweitert…“ klicken und dann „Risiko akzeptieren und fortfahren“:





3.1.1 Swagger

Es erscheint folgendes „Swagger“-Fenster:



Damit kann man die Controller-Methoden sofort ausprobieren – die Bedienung ist sehr intuitiv.

3.2 dotnet watch run

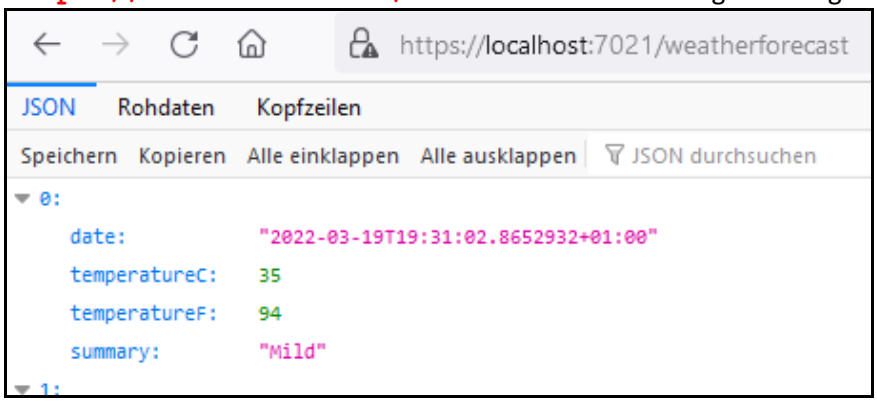
Das Projekt kann auch mit **`dotnet watch run`** gestartet werden:

```
C:\_PR\CSharp\PR4\300-399\WebApiDemo\WebApiDemo>dotnet watch run
watch : Hot reload enabled. For a list of supported edits, see https://aka.ms/dotnet/hot-reload. Pr
watch : Building...
  Determining projects to restore...
  All projects are up-to-date for restore.
  WebApiDemo -> C:\_PR\CSharp\PR4\300-399\WebApiDemo\WebApiDemo\bin\Debug\net6.0\WebApiDemo.dll
watch : Started
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7021
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5021
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

Achtung: die Ports werden zufällig vergeben und können daher auf anderen PCs andere sein. Die „http“-Version ist immer ein 5000er Port, die „https“-Version ein 7000er Port.

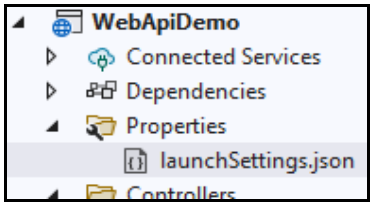
3.2.1 Testen im Browser

Man kann auch direkt im Browser testen. Ohne weitere Änderungen liefert
`http://localhost:5021/weatherforecast` bzw.
`https://localhost:7021/weatherforecast` folgendes Ergebnis:



3.2.2 Launch settings

Die URL bzw. den Port, mit dem der Webserver startet, kann man in Properties → **`launchSettings.json`** einstellen:



Diese Datei sieht zu Beginn so aus:

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5604",
      "sslPort": 44307
    }
  },
  "profiles": {
    "WebApiDemo": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7021;http://localhost:5021",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Am besten den Teil mit iisSettings löschen:

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "WebApiDemo": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7021;http://localhost:5021",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Bei Bedarf dort dann die „launchURL“ anpassen:

```
"WebApiDemo": {
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "weatherforecast",
  "applicationUrl": "https://localhost:5001;http://localhost:5000",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Bei Problemen mit https kann man diese URL auch aus applicationUrl löschen.

4 Programmstruktur

Ein kurzer Blick soll noch auf das eigentliche Framework geworfen werden.

4.1 Program.cs

Das Programm ist eine Konsolenapplikation, die Methode Main fehlt aber – Program.cs wird mit Top Level Statements notiert.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Dependency Injection

Middleware

Hier wird das Framework konfiguriert. Darin werden einerseits die benötigten Klassen/Services mit Dependency Injection zur registriert, andererseits die sog. Middleware aufgebaut.

4.1.1 Dependency Injection

Darin werden die Services als Singletons registriert. Sie besteht zu Beginn nur aus einer Codezeile, nämlich dem Registrieren des MVC-Moduls.
In dieser Klasse werden später eigene Services, wie z.B. der Datenbank-Kontext eingetragen.

4.1.2 Middleware

Hier wird festgelegt, welche Module in welcher Reihenfolge die Webrequests behandeln.

4.2 appsettings.json

Diese Datei ist für Benutzereinstellungen gedacht. Vorerst sind nur LogLevel definiert.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Viele Logs machen die Ausgabe oft sehr unübersichtlich, es bietet sich an, die vielen automatischen Logs von Microsoft auszuschalten.
Achtung: es gibt zwei Dateien – appsettings.json und appsettings.Development.json

appsettings.json

Schema: http://json.schemastore.org/appsettings

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Warning"
7      }
8    },
9    "AllowedHosts": "*"
10 }
```

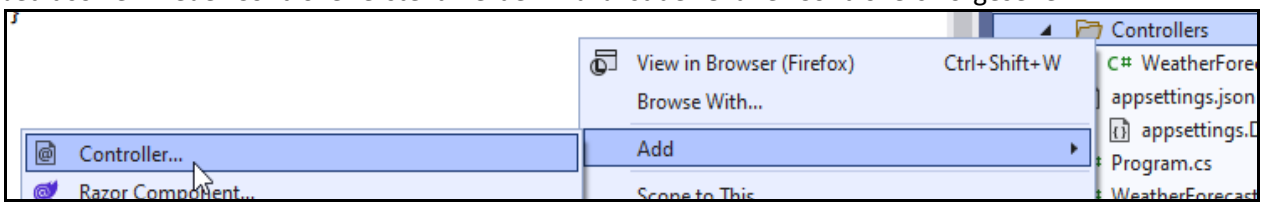
appsettings.Development.json

Schema: <No Schema Selected>

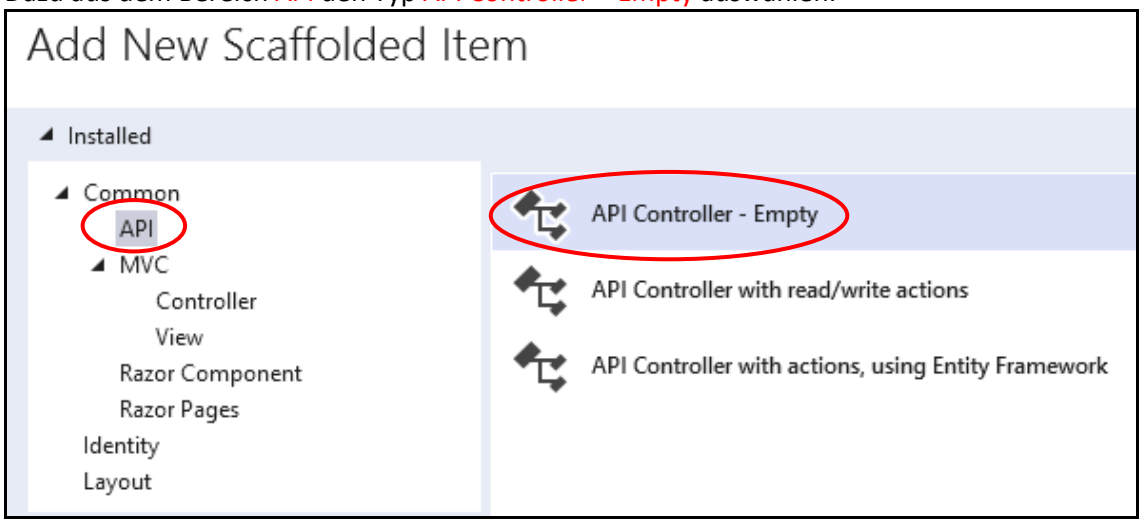
```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Warning"
7      }
8    }
9  }
```

5 Neuer Controller

Jetzt soll ein neuer Controller erstellt werden. Dazu ist der Ordner Controllers vorgesehen.



Dazu aus dem Bereich API den Typ API Controller – Empty auswählen:



Der Name des Controllers ist dann grundsätzlich frei zu vergeben, es ist aber üblich, das Suffix Controller zu benutzen, z.B. ValuesController.
Die Struktur ist so wie von bereits vorhandenen WeatherForecastController bekannt:

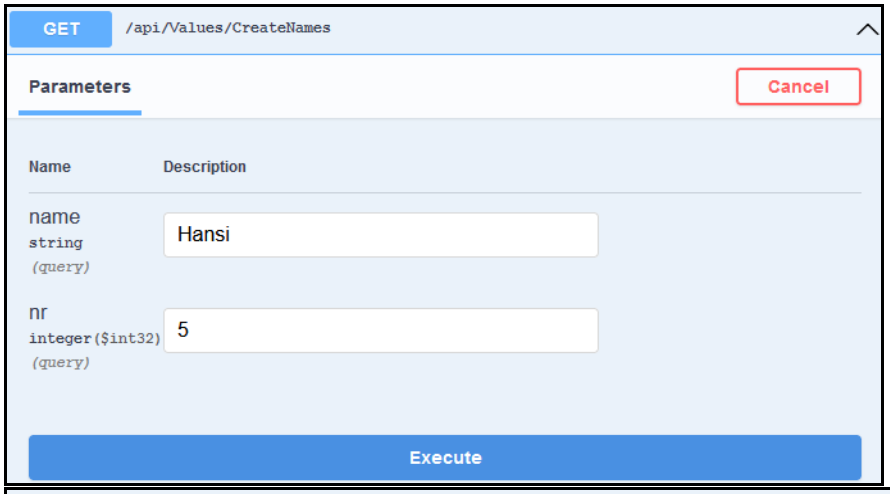
```
namespace WebApiDemo.Controllers;

[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
}
```

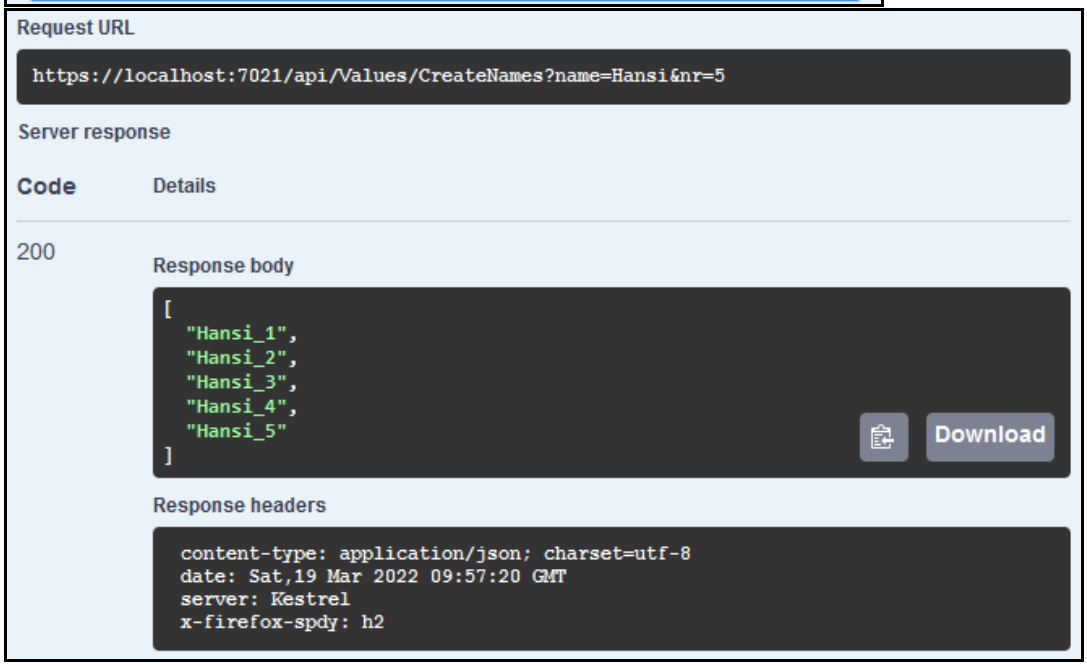
5.1 Action-Methode

Eine Methode, die Daten liefert, gibt einfach ganz normale C#-Objekte zurück. Dabei sollten immer Objekte definierter Klassen retourniert werden (und keine anonymen Objekte).
Die Methode muss mit [HttpGet] markiert werden, ansonsten kann man sie nicht von einer normalen C#-Methoden unterscheiden:

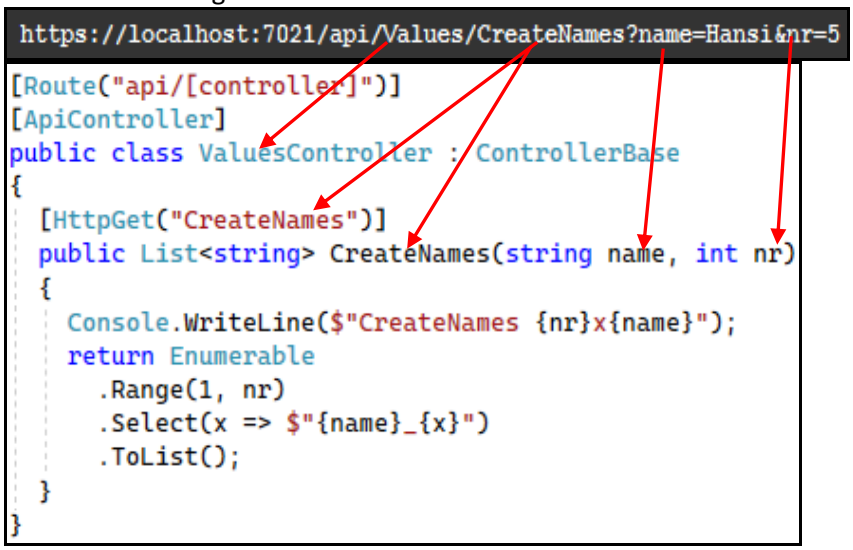
```
[HttpGet("CreateNames")]
public List<string> CreateNames(string name, int nr)
{
  Console.WriteLine($"CreateNames {nr}x{name}");
  return Enumerable
    .Range(1, nr)
    .Select(x => $"{name}_{x}")
    .ToList();
}
```

Aus diesem Objekt wird anhand der Properties automatisch in ein JSON-Objekt erzeugt.
Man kann das sofort im Browser mit Swagger testen.



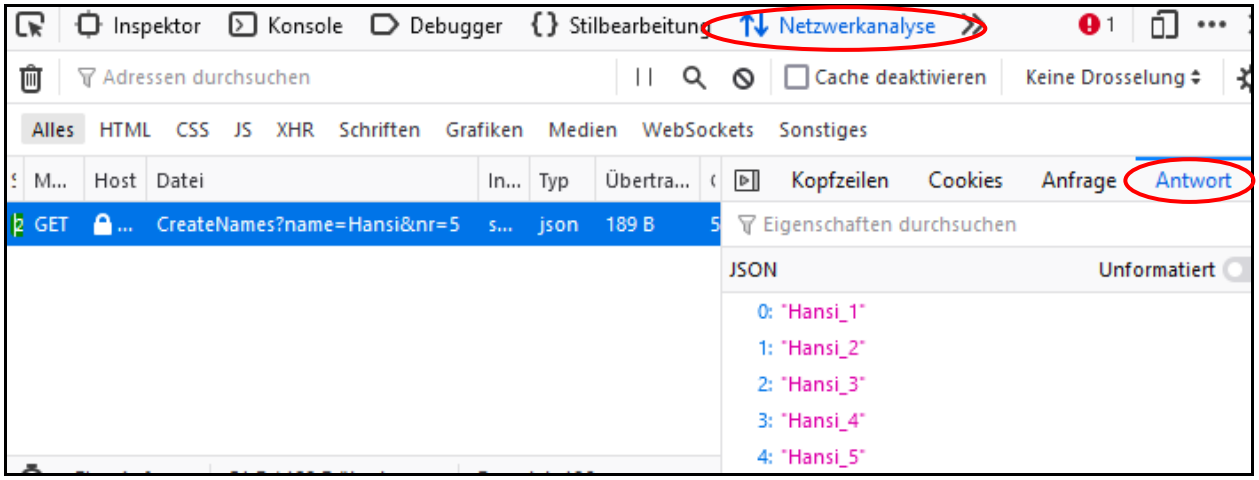


Die Parameter des Requests werden als Query-Parameter an die URL angehängt und auf die Parameter der Methode übertragen.



5.1.1 Netzwerkanalyse

Zum Testen bietet sich immer der Browser an, indem man in der Developerkonsole den Netzwerkverkehr betrachtet.


Inspektor
Konsole
Debugger
Stilbearbeitung
Netzwerkanalyse
1
Adressen durchsuchen
Cache deaktivieren
Keine Drosselung
Alles
HTML
CSS
JS
XHR
Schriften
Grafiken
Medien
WebSockets
Sonstiges
M...
Host
Datei
In...
Typ
Übertra...
Kopfzeilen
Cookies
Anfrage
Antwort
GET
CreateNames?name=Hansi&nr=5
s...
json
189 B
Eigenschaften durchsuchen
JSON
Unformatiert
0: "Hansi_1"
1: "Hansi_2"
2: "Hansi_3"
3: "Hansi_4"
4: "Hansi_5"

6 Attribute Routing

Bei Webservices ist wesentlich, wie eine Browser-Request zu einer Methode in einem Controller findet sowie die zusätzlichen Daten auf Methodenparameter schreibt. Diese Regel kann konfiguriert werden und nennt sich Attribute-Routing. Dabei schreibt man die Routendefinition direkt über die Klasse bzw. über die Methode.
Dabei kann man für Klassen- u. Methodenname Platzhalter verwenden:

- **[controller]**: entspricht dem aktuellen Namen der Controller-Klasse
- **[action]**: entspricht dem Namen der Methode
- **{xxx}**: entspricht einem symbolischen Namen

6.1 [Route] – Default Route

Schreibt man eine Route über den Klassennamen des Controllers, so gilt diese Route als Basis für alle Methoden dieses Controllers. Alle Routenparameter bei HttpGet werden dann an diese Route einfach angehängt.
Ausnahme: startet bei HttpGet die Route mit „/“, so wird die Default-Route ignoriert und kann bzw. muss die gesamte Route angeben.
Als Basis soll der Controller DemoController dienen:

```
[Route("[controller]")]
[ApiController]
public class DemoController : ControllerBase
{
}
```

Hinweis: in diesem Fall habe ich das Präfix „api/“ entfernt.
Hier ein paar Beispiele:

`[HttpGet("ActionX")]`
`public string ActionX()`
`{`
`  return BuildMethodString();`
`}`
	Der Name bei HttpGet wird and die Default-Route angehängt:
	`Demo/ActionX`

`[HttpGet("ActionY")]`
`public string Quaxi()`
`{`
`  return BuildMethodString();`
`}`
	Es zählt die Name bei HttpGet, nicht der Methodenname:
	`Demo/ActionY`

`[HttpGet("[action]")]`
`public string ActionZ()`
`{`
`  return BuildMethodString();`
`}`
	Der Platzhalter [action] steht für den aktuellen Methodennnamen:
	`Demo/ActionZ`

`[HttpGet("/Dummy/SomeAction")]`
`public string BlaBla()`
`{`
`  return BuildMethodString();`
`}`
	Man kann die Controller-Default-Route jederzeit komplett überschreiben, indem man die Action-Route mit einem „**/**“ beginnt:
	`Dummy/SomeAction`

`[HttpGet("/api/[controller]/[action]")]`
`public string GetAll()`
`{`
`  return BuildMethodString();`
`}`
	Diese Varianten kann man auch mischen:
	`api/Demo/GetAll`

6.1.1 Änderung Default Route

Häufig wird die Default-Route auf [controller]/[action] geändert.

```
[Route("[controller]/[action]")]
[ApiController]
public class OtherDemoController : ControllerBase
```

Das würde sich so auswirken:

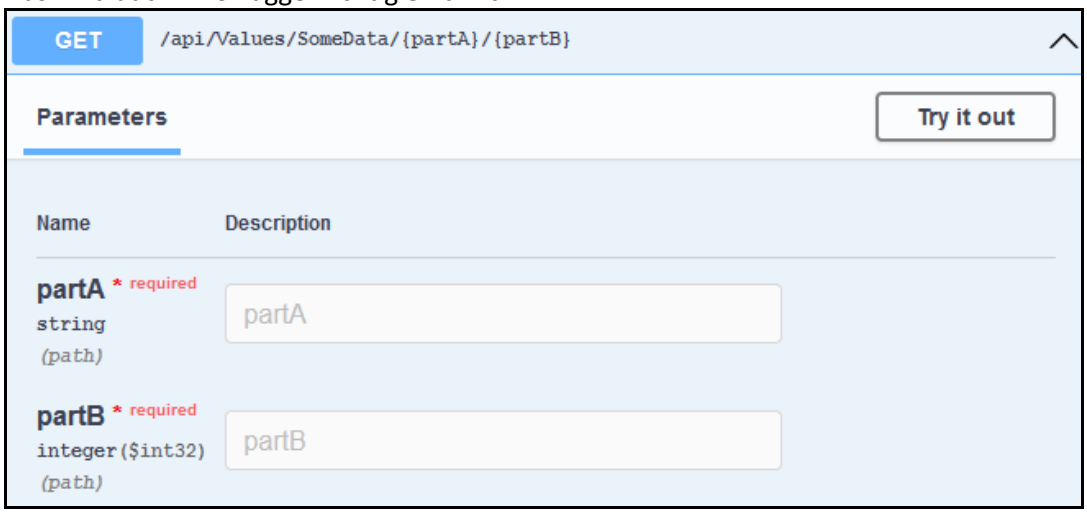
`[HttpGet] public string ActionX() => BuildMethodString();`	**`OtherDemo/ActionX`**
`[HttpGet] public string Quaxi() => BuildMethodString();`	**`OtherDemo/Quaxi`**
`[HttpGet("[action]")] public string ActionZ() => BuildMethodString();`	**`OtherDemo/ActionZ/ActionZ`**

6.2 Routenparameter

Parameter können wie weiter oben gezeigt immer als Queryparamter übergeben werden. Häufig werden aber Parameter als Teil der Route definiert. Dabei werden die dynamischen Parameter in { } notiert.
Beispiel im ValuesController:

```
[HttpGet("[action]/{partA}/{partB}")]
public string SomeData(string partA, int partB)
{
    return $"SomeData for {partA} {partB}";
}
```

Das entspricht der Route z.B. ***api/Values/SomeData/abc/123***
Das wird auch im Swagger richtig erkannt:

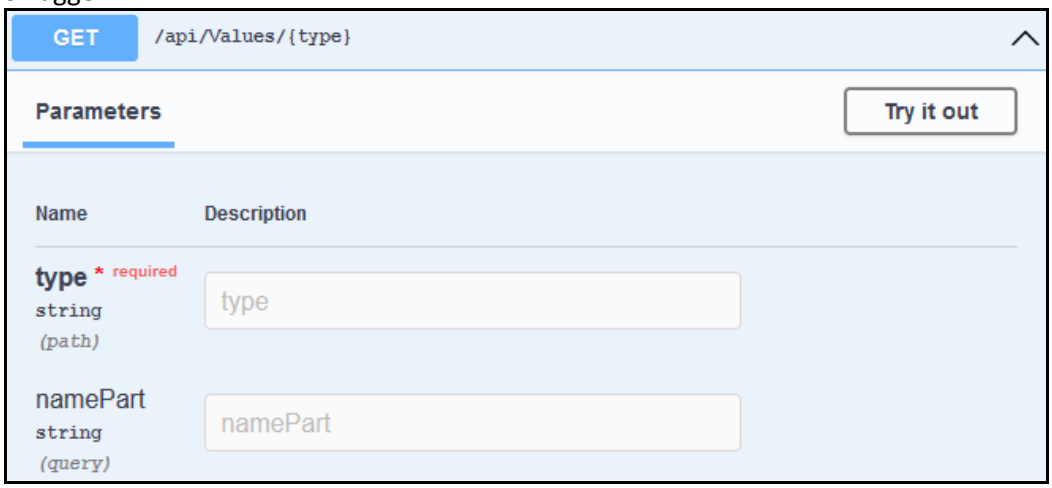


6.2.1 Queryparameter

Man kann Routen- u. Queryparameter auch mischen.

```
[HttpGet("{type}")]
public List<string> FindPersons(string type, string namePart)
{
    Console.WriteLine($"type={type} / namePart={namePart}");
    return new();
}
```

Swagger:



Das wird dann in eine entsprechende URL umgesetzt:

```
https://localhost:7021/api/Values/persons?namePart=au
```

7 Datenbank

Eine Datenbank wird wie gewohnt benutzt: Wie in den entsprechenden Dokumenten als „Database First“ scaffollden oder als „Code First“ entwerfen.

7.1 Dependency Injection

In beiden Fällen ist es ein absolutes NoGo, die Datenbank im Controller mit
`var db = new NorthwindContext(); //Never do this!!!!!`
zu erzeugen.
Das muss immer mit Dependency Injection erfolgen. Wie weiter oben besprochen ist dafür ein Bereich in Program.cs vorgesehen:

```
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddDbContext<NorthwindContext>();
```

7.1.1 Code First – EnsureCreated

Manchmal möchte man gleich zu Beginn sicherstellen, dass eine Code First Datenbank erstellt wurde bzw. mit Seed Daten eintragen.
Das geht mit folgender Code-Sequenz:

```
var app = builder.Build();

Console.WriteLine("Creating StudentCourseContext");
var scope = app.Services.CreateScope();
var studentCourseDb = scope.ServiceProvider.GetRequiredService<StudentCourseContext>();
studentCourseDb.Database.EnsureDeleted();
studentCourseDb.Database.EnsureCreated();
```

Hinweis 1: dazu muss natürlich die Datenbank vorher mit **`AddDbContext`** registriert worden sein.
Hinweis 2: es gibt eine bessere Möglichkeit, das zu tun (Details dazu später).

7.2 Verwenden

Die Verwendung erfolgt dann eben dadurch, dass man die Referenz im Konstruktor anfordert.
Dazu einen neuen Controller NorthwindController erstellen und den Konstruktor wie folgt ändern:

```
[Route("[controller]")]
[ApiController]
public class NorthwindController : ControllerBase
{
    private readonly NorthwindContext _db;

    public NorthwindController(NorthwindContext db) => _db = db;
}
```

Eine Methode zum Lesen aller Categories könnte dann so aussehen:

```
[HttpGet("Categories")]
public List<Category> GetAllCategories()
{
    return _db.Categories.ToList();
}
```

7.3 appsettings.json

Es wird empfohlen, nicht den Default-Konstruktor und damit den Fallback-Mechanismus anzuwenden, sondern den Konstruktor mit den Optionen zu benutzen. Dabei sollte man den Connectionstring nicht direkt angeben, sondern aus **`appsettings.json`** (bzw. **`appsettings.Development.json`**) auslesen. Neben den bereits vorhandenen Einstellungen kann man auch beliebige andere Eigenschaften für die App definieren.

```
},
"AllowedHosts": "*",
"ConnectionStrings": {
  "Northwind": "server=(LocalDB)\\mssqllocaldb; attachdbfilename=D:\\Temp\\Nor
}
```

Nennt man einen Eintrag **ConnectionStrings** kann dieser mit **builder.Configuration.GetConnectionString** ausgelesen werden:

```
string connectionString = builder.Configuration.GetConnectionString("Northwind");
Console.WriteLine($"******* ConnectionString: {connectionString}");
builder.Services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString));
```

Da eine der häufigsten Fehlerquellen bei Projekten ein Fehler im Bereich des Connectionstrings liegt, empfehle ich dringend, diesen zu Beginn auf die Konsole auszugeben!

7.3.1 Relativer Pfad - |DataDirectory|

Etwas lästig ist noch, dass der Pfad der Datenbank als absoluter Pfad eingetragen ist. Besser wäre, einen relativen Pfad mit |DataDirectory| angeben zu können.

```
"ConnectionStrings": {
  "Northwind": "server=(LocalDB)\\mssqllocaldb; attachdbfilename=|DataDirectory|\\Northwnd.mdf;i
  "Northwind_": "server=(LocalDB)\\mssqllocaldb; attachdbfilename=C:\\Temp\\Northwnd.mdf;integra
}
```

|DataDirectory| wird von EF Core auf jenen Pfad übersetzt, in dem das aktuelle exe liegt, also **../bin/debug/net6.0**. Nicht vergessen, in diesem Fall die Datei Northwnd.mdf auf „Copy if newer“ zu stellen!

8 DTOs

Es sollen wie bereits mehrmals besprochen nie die Datenbank-Objekte direkt retourniert werden, sondern immer sogenannte **D**ata **T**ransfer **O**bjects, kurz **DTO**.

8.1 Motivation

Absolutes „No Go“ ist z.B. Datenbankklassen zurückzugeben. Zur Demonstration, warum das manchmal mehr ist als ein „nice to have“, zeigt folgendes Beispiel:

```
[HttpGet("Products/{id}")]
public Product? GetProduct(int id)
{
  Console.WriteLine($"Northwind GetProduct {id}");
  return _db.Products
    .Include(x => x.Category)
    .SingleOrDefault(x => x.ProductId == id);
}
```

Damit bekommt man folgende Fehlermeldung:

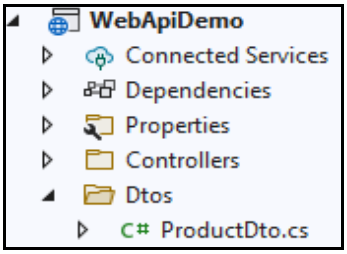
```
fail: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware[1]
      An unhandled exception has occurred while executing the request.
      System.Text.Json.JsonException: A possible object cycle was detected. This can either be due to a c
Category.Products.Category.Products.Category.Products.Category.CategoryId.
         at System.Text.Json.ThrowHelper.ThrowJsonException_SerializerCycleDetected(Int32 maxDepth)
         at System.Text.Json.Serialization.JsonConverter`1.TryWrite(Utf8JsonWriter writer, T& value, Json
```

Das Problem ist die in der Fehlermeldung erwähnte „**self referencing loop**“. Um ein Product zu serialisieren werden alle Properties in einen JSON-String umgewandelt, also auch die Property Category. Die Klasse Category hat aber wiederum eine Referenz auf eine Liste von Products. Um eine Category serialisieren zu können müssen also die Products serialisiert werden und so beißt sich die Katze in den Schwanz.

8.2 Dto

Man erstellt also einfache Klassen, die nur eine Ansammlung von Properties sind. Oftmals sind diese fast identisch zu einer Datenbank-Klasse, wobei man aber nur jene Properties angibt, die auch tatsächlich im Frontend benötigt werden. Üblicherweise erfolgt das in einem Ordner namens Dtos:

```
public class ProductDto
{
  public int ProductId { get; set; }
  public string ProductName { get; set; } = "";
  public decimal? UnitPrice { get; set; }
  public string CategoryName { get; set; } = "";
}
```



Es gibt keinen Grund, Properties umzubenennen, wenn sich die Bedeutung nicht ändert. Daraus ergibt sich außerdem ein entscheidender Vorteil, wie wir weiter unten noch sehen werden.

Die Action-Methode würde dann so aussehen:

```
[HttpGet("Products/{id}")]
public ProductDto? GetProduct(int id)
{
  Console.WriteLine($"Northwind GetProduct {id}");
  return _db.Products
    .Include(x => x.Category)
    .Select(x => new ProductDto
    {
      ProductId = x.ProductId,
      ProductName = x.ProductName,
      UnitPrice = x.UnitPrice,
      CategoryName = x.Category!.CategoryName
    })
    .SingleOrDefault(x => x.ProductId == id);
}
```

Response body

```
{
  "productId": 1,
  "productName": "Chai",
  "unitPrice": 18,
  "categoryName": "Beverages"
}
```

Man beachte, dass die Konventionen für Großschreibung in C# und Kleinschreibung in Javascript automatisch eingehalten werden. D.h. die großgeschriebenen Properties werden ohne weitere Einstellung in **kleingeschriebene** Properties des JSON-Objekts umgewandelt.