

.Net Core WebApi Services

1 EINFÜHRUNG	3
1.1 Project create script	3
1.1.1 Gitlab	3
1.1.2 Ausführen	3
1.1.3 Testen	4
2 REST API	5
2.1 Controller Actions	5
2.2 DTOs	6
3 REST CLIENT – VISUAL STUDIO CODE PLUGIN	7
3.1 Get	7
3.2 Post	8
3.3 Variable	8
3.4 _requests.http	9
4 SERVICE	10
4.1 Service erstellen	10
4.2 Service verwenden	10
4.3 Methoden verschieben	11
4.4 DTO Verwendung vereinfachen	11
4.5 Testen	12
4.6 Interface pro Service	12
4.7 Request vs. Reply	13
4.8 Post	13
4.8.1 Controller	13
4.8.2 Service	13
4.9 FromQuery/FromBody	13
4.9.1 FromQuery	14
4.9.2 FromBody	14
4.10 ResultFilter	14
4.10.1 Synchron	14
4.10.2 Async	15
4.10.3 Verwenden	15
5 BACKGROUND TASKS	16
5.1 BackgroundService	16
5.2 Starten	16
5.3 Synchron vs. Asynchron	16
5.4 Datenbank erstellen	17
6 STATUSCODE / ACTIONRESULT	18
6.1 StatusCode	18
6.2 ActionResult<T>	18
6.2.1 CreatedAtAction	18
6.2.2 ActionResult	19
7 FEHLERBEHANDLUNG	20
7.1 Testen	20
7.2 Exception	20
7.3 Globaler Exceptionhandler	20

7.4 Model Validation	21
8 ZUSAMMENFASSUNG	22
9 UNIT TEST	23
9.1 UnitTest-Klasse	23
9.2 Testmethoden	23
10 SWAGGER/OPENAPI	25
10.1 API checken	25
10.1.1 Route testen	25
10.1.2 swagger.json	26
10.1.3 Schemas	27
10.2 ActionResult<T>	28

1 Einführung

Es gibt viele unterschiedliche Varianten, wie man ein „Best Practice“ für WebApi lösen könnte, d.h. das folgende Dokument ist nur als eine dieser Varianten zu sehen.

1.1 Project create script

Für die Erstellung eines WebApi-Projekts sind immer wieder dieselben Schritte notwendig. Diese kann man auch von der Konsole aus ausführen.

Zur Erleichterung habe ich ein kleines Batch-Script erstellt, mit dem man diese Schritte automatisch ausführen und somit von der Konsole aus ein Projekt erstellen und auch gleich testen kann.

1.1.1 Gitlab

Dieses Script liegt in Gitlab unter <https://gitlab.com/rgrueneis/createwebapiproject>. Dort ist auch eine kurze Beschreibung vorhanden, wie man das Script verwendet.

Man klonst sich dieses Tool mit folgendem Befehl:

```
git clone https://gitlab.com/rgrueneis/createwebapiproject.git
```

1.1.2 Ausführen

Der Befehl lautet dann createWebapiProject. Bei Aufruf ohne Parameter werden die Optionen erklärt.

```
D:\Temp\createwebapiproject>createWebapiProject
usage: createWebapiProject solutionName [pathToDbFile] [tableName] [targetFolder]
  solutionName: Name of the solution and name of folder
  pathToDbFile: If set, an Entity data model will be generated. Database has to reside in local folder or absolute path given.
  tableName:    For this table a Get-WebService is generated. Default: Categories
  targetFolder: Folder where to generate the project to. Default: Subfolder with solution name in current folder
```

- das Projekt wird wie die Solution benannt.
- als Port wird immer 5000 für http bzw. 5001 für https verwendet.
- Benötigt man im Projekt eine Datenbank, kann der Pfad zur MDF-Datei (oder auch zu einer Sqlite-Datei) angegeben werden, und es wird automatisch das Entity Data Model erstellt.
- Da ein Controller erzeugt wird, das aus einer Tabelle Daten liefert, muss der Name der Tabelle angegeben werden (case sensitive)
- Als vierten Parameter kann man ein Verzeichnis angeben, in das das Projekt erstellt wird. Leerzeichen sind dabei nicht erlaubt.

Aufruf z.B.: **createWebapiProject WebApiDemo Northwnd.mdf Products D:\Temp**

Die Ausgabe müsste so aussehen:

```
D:\Temp\createwebapiproject>createWebapiProject WebApiDemo Northwnd.mdf Products D:\Temp
-----
.Net Core WebApi project create script
v6.1.0, 2022-03-30
(C)Robert Grueneis/HTL Grieskirchen
-----
Current dotnet version:
6.0.201
Current dotnet-ef version:
Entity Framework Core .NET Command-line Tools
6.0.0
-----
Using .Net Core:      6.0
Using Entity Framework: 6.0.0
-----
solution      --> WebApiDemo
project       --> WebApiDemo
database      --> Northwnd.mdf
target folder --> D:\Temp\WebApiDemo
-----
Database name = Northwind
fullDbPath = D:\Temp\createwebapiproject\Northwnd.mdf
Creating new solution WebApiDemo
Die Vorlage "Projektmappendatei" wurde erfolgreich erstellt.
Creating new WebApi Project WebApiDemo
```

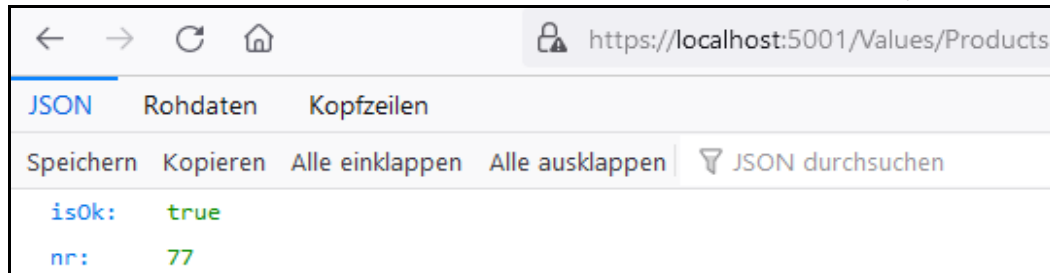
```
...
Writing replaced file to Properties\launchsettings.json
ExtensionMethods.cs
D:\Temp\createwebapiproject\_templates\ExtensionMethods.cs: <$$PROJECT$$> --> <WebApiDemo> (1 times)
Writing replaced file to ExtensionMethods.cs
GlobalUsings.cs
D:\Temp\createwebapiproject\_templates\GlobalUsings.cs: <$$PROJECT$$> --> <WebApiDemo> (1 times)
D:\Temp\createwebapiproject\_templates\GlobalUsings.cs: <$$DBPROJECT$$> --> <WebApiDemoDb> (1 times)
Writing replaced file to GlobalUsings.cs
-----
Installation finished
-----
Press a key to start the project and open browser with url http://localhost:5000/Values/Products ...
```

Drückt man dann eine Taste, wird ein neues Fenster geöffnet, in dem der Server gestartet wird:

```
watch : Building...
WebApiDemoDb -> D:\Temp\WebApiDemo\WebApiDemoDb\bin\Debug\net6.0\WebApiDemoDb.dll
WebApiDemo -> D:\Temp\WebApiDemo\WebApiDemo\bin\Debug\net6.0\WebApiDemo.dll
watch : Started
***** ConnectionString: Server=(LocalDB)\mssqllocaldb;attachdbfilename=D:\Temp\createwebapiproject\North
***** Swagger enabled: http://localhost:5000/swagger (to set as default route: see launchsettings.json)
12:52:15 GetProducts
```

1.1.3 Testen

Gibt man dann im Browser **localhost:5000/Values/Products** ein, müsste es so aussehen:



Hinweis: wird keine Datenbank verwendet, lautet die Url unverändert localhost:5000/Values

2 Rest Api

Zum Vergleich: <https://restfulapi.net/http-methods/>

Method	CRUD	Entire Collection (e.g. /users)	Specific Item (e.g. /users/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID.	Avoid using POST on single resource
GET	Read	200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single user. 404 (Not Found), if ID not found or invalid.
PUT	Update/ Replace	404 (Not Found), unless you want to update every resource in the entire collection of resource.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
PATCH	Partial Update/ Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
DELETE	Delete	404 (Not Found), unless you want to delete the whole collection — use with caution.	200 (OK). 404 (Not Found), if ID not found or invalid.

Das heißt also, wir brauchen folgende Signaturen:

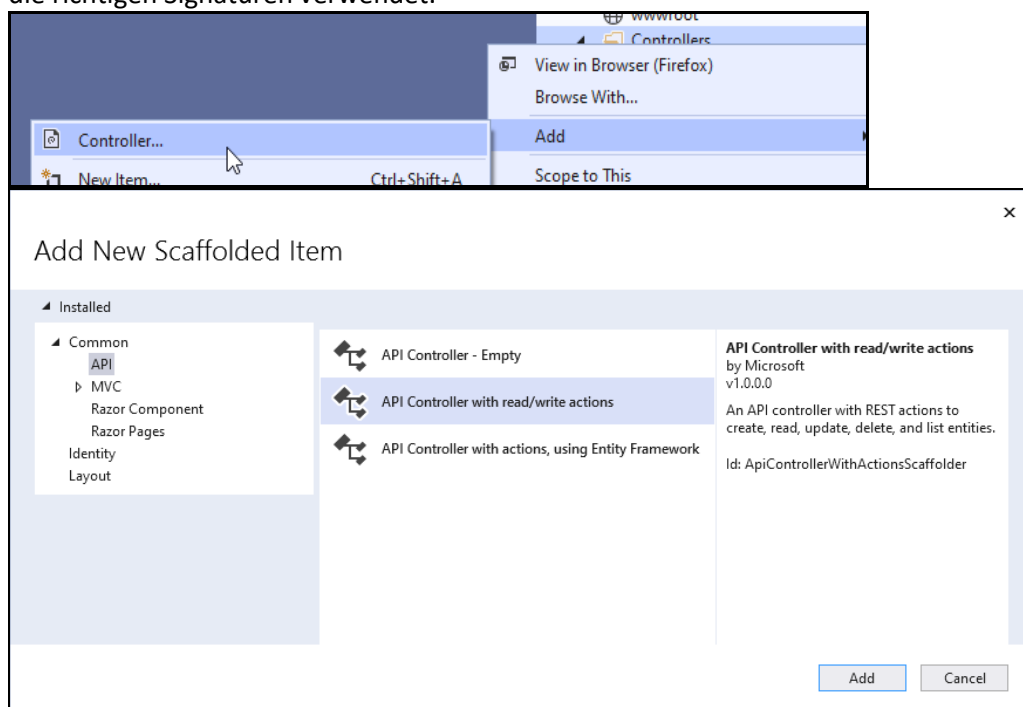
Verb	Url	body	Reply
Get	Categories		List<Category>
Get	Categories/{id}		Category
Post	Categories	Category-Objekt	Category mit aktueller Id
Put	Categories/{id}	Category-Objekt	Category mit Änderungen
Delete	Categories/{id}		Gelöschtes Category-Objekt

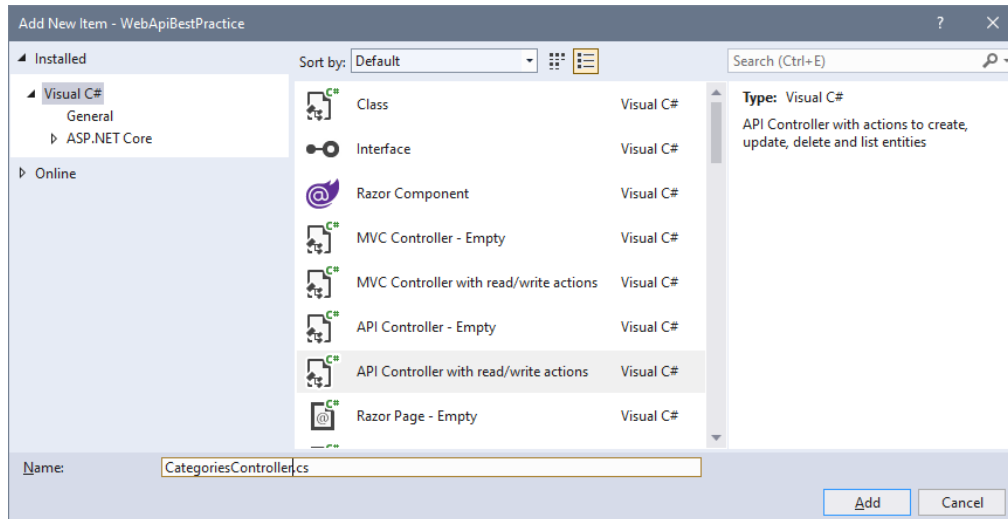
Die Spezifikation kann man unter <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html> nachlesen.

2.1 Controller Actions

Im generierten Code wird als Basisroute `[Route("api/[controller]")]` verwendet. Ob man das Suffix **api** lässt, oder es entfernt ist Geschmackssache. Ich lösche es üblicherweise (außer es wird zusätzlich ein SignalR-Hub verwendet), weil das Backend ohnehin nur API-calls bedient und daher die Angabe meiner Meinung nach redundant ist.

Für die Categories soll ein neuer Controller erstellt werden, der GET, POST, PUT und DELETE ermöglicht und dabei die richtigen Signaturen verwendet.





Dadurch hat man schon einmal mehr oder weniger die richtigen Signaturen (werden für String generiert – durch Category ersetzen).

```
[Route("[controller]")]
[ApiController]
public class CategoriesController : ControllerBase
{
    [HttpGet]
    public IEnumerable<Category> Get()...

    [HttpGet("{id}")]
    public Category Get(int id)...

    [HttpPost]
    public Category Post([FromBody] Category category)...

    [HttpPut("{id}")]
    public Category Put(int id, [FromBody] Category category)...

    [HttpDelete("{id}")]
    public Category Delete(int id)...
}
```

Die jeweiligen HTTP Verbs des Rest-API werden über Attribute wie HttpGet, HttpPost, ... angegeben. Wird ein einzelnes Objekt angefragt, wird dessen Id als Teil der Route notiert, daher also z.B. `HttpPut("{id}")`.

2.2 DTOs

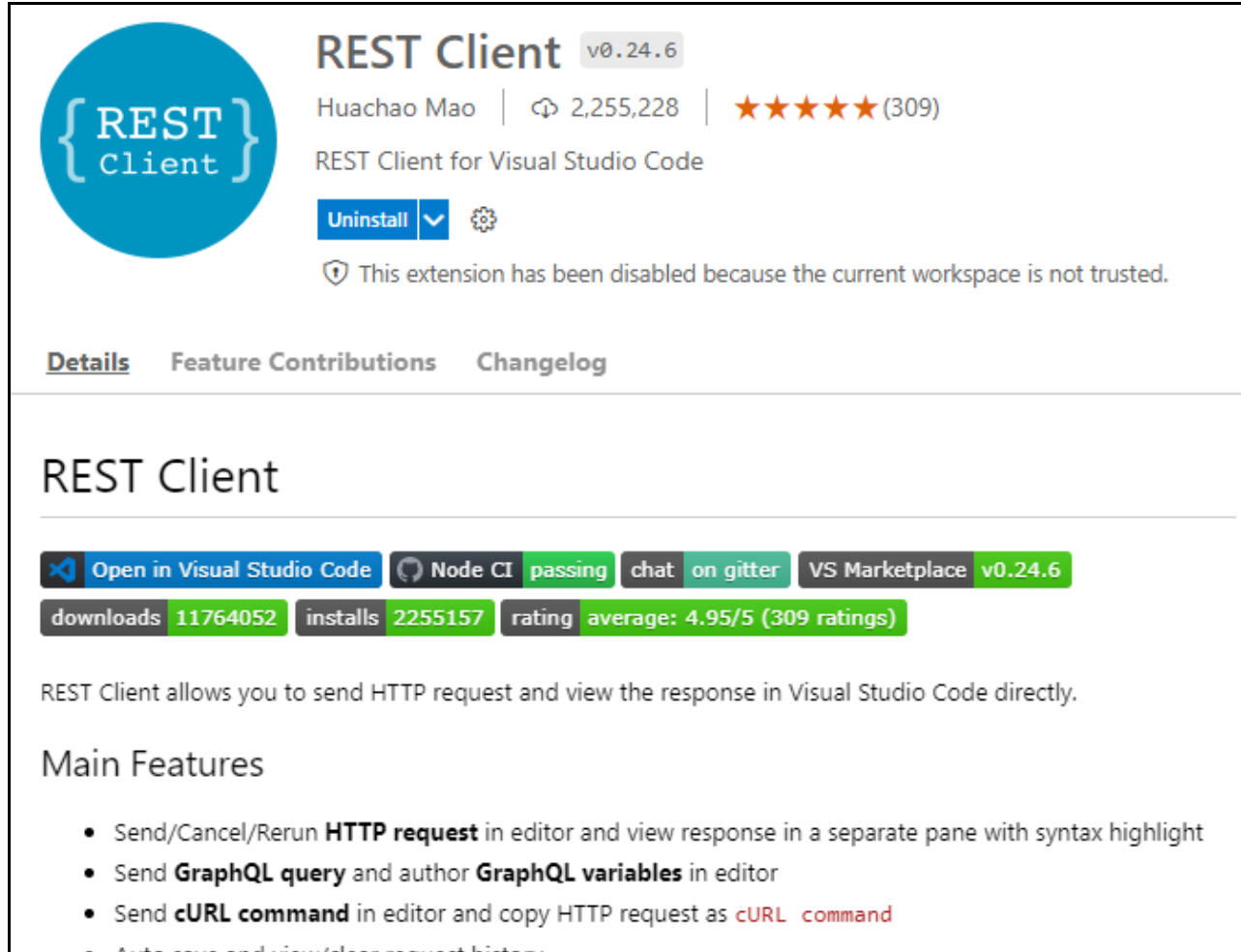
Es gilt nach wie vor, dass keine Datenbank-Objekte zum Frontend übertragen werden sollen. Wie im letzten Tutorial besprochen also immer DTOs erzeugen, also z.B.:

```
public class CategoryDto
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; } = "";
    public string Description { get; set; } = "";
}
```

3 REST Client – Visual Studio Code plugin

Die Verwendung von Swagger ist zwar einfach, manchmal aber doch etwas umständlich, weil man mehrere Klicks und evtl. Einträge in Textboxen braucht, um den Request abzuschicken.

Neben den vielleicht bekannten Tools wie CURL oder PostMan für Visual Studio Code ein sehr einfaches und übersichtlich zu bedienendes Plugin „REST Client“ mit dem man Webrequests in einer Datei mit dem Projekt speichern und innerhalb von Visual Studio Code ausführen kann.



The screenshot shows the REST Client extension page in the Visual Studio Code Marketplace. The extension is by Huachao Mao, has 2,255,228 downloads, and a 5-star rating from 309 reviews. It is currently disabled because the workspace is not trusted. The page includes tabs for Details, Feature Contributions, and Changelog. The main features listed are: Send/Cancel/Rerun HTTP request in editor and view response in a separate pane with syntax highlight; Send GraphQL query and author GraphQL variables in editor; Send cURL command in editor and copy HTTP request as cURL command; and Auto save and view/clear request history.

Die wichtigsten Punkte:

- Man schreibt die Requests einfach in eine Datei mit Extension **http**.
- Die Ergebnisse der Requests werden direkt in Visual Studio Code angezeigt.
- Die einzelnen Request werden mit (mindestens 3) **###** getrennt.

3.1 Get

```
Send Request
GET http://localhost:5000/categories HTTP/1.1
###
Send Request
GET http://localhost:5000/categories/1 HTTP/1.1
###
```

Bei Klick auf „Send Request“ wird der entsprechende Request abgeschickt und das Ergebnis in einem eigenen Fenster angezeigt.

<pre>##### Categories Send Request GET http://localhost:5000/categories HTTP/1.1 #### Send Request GET http://localhost:5000/categories/1 HTTP/1.1 ####</pre>	<pre>1 HTTP/1.1 200 OK 2 Connection: close 3 Date: Sat, 16 May 2020 07:43:02 GMT 4 Content-Type: application/json; charset=utf-8 5 Server: Kestrel 6 Transfer-Encoding: chunked 7 8 { 9 "categoryId": 1, 10 "categoryName": "Beverages", 11 "description": "Soft drinks, coffees, teas, beer 12 s, and ales", 13 "picture": "FRwvAAIAAANA4AFAAhAP///9CaXRtYXAg 14 SW1hZ2UuIGFEnbnQuIGlIdHVvZ0ABBOAAAgAAAcAAABOQnJ1c2</pre>
---	---

3.2 Post

Ein POST-Request funktioniert ähnlich. In der Zeile unterhalb gibt man den Content-Type als **application/json**, mit einer Leerzeile getrennt folgen dann die Daten in JSON-Notation:

<pre>POST http://localhost:5000/categories HTTP/1.1 Content-Type: application/json { "categoryName": "Vegetables", "description": "all kinds of vegetables" } ####</pre>	<pre>1 HTTP/1.1 201 Created 2 Connection: close 3 Date: Fri, 15 May 2020 14:17:38 GMT 4 Content-Type: application/json; charset=utf-8 5 Server: Kestrel 6 Transfer-Encoding: chunked 7 Location: http://localhost:5000/Categories 8 9 { 10 "categoryId": 3011, 11 "categoryName": "Vegetables", 12 "description": "all kinds of vegetables", 13 "picture": null, 14 "products": [] 15 }</pre>
---	--

Die Art des Rückgabeobjekts hängt natürlich von der Implementierung ab.

3.3 Variable

Um längere URLs oder Parameter nicht immer wiederholen zu müssen, kann man diese in Variablen speichern.

```
1 reference
@hostname = localhost
1 reference
@port = 5000
2 references
@url = http://{{hostname}}:{{port}}
6 references
@categories = {{url}}/categories
0 references
@products = {{url}}/products
2 references
@categoryId = 5012
##### Categories
Send Request
GET {{categories}} HTTP/1.1
####
Send Request
GET {{categories}}/1 HTTP/1.1
####
```


3.4 _requests.http

Die Requests können in jeder beliebigen Datei mit Dateiendung **.http** gespeichert werden. Idealerweise vergibt man einen Namen, den man bei jedem Projekt verwendet, z.B. **_requests.http**. Stellt man dafür noch ein, dass diese Extension immer mit Visual Studio Code geöffnet wird, kann man sehr schnell das Backend testen (auch ohne ein entsprechendes jQuery oder Angular Projekt zu haben).

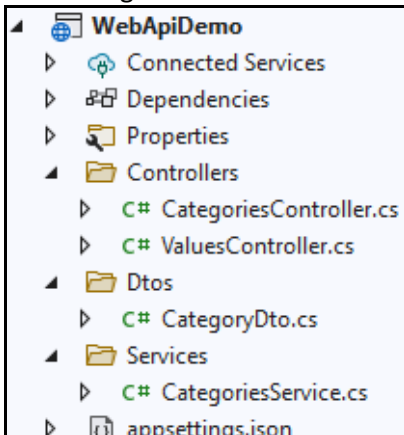
4 Service

Es ist üblich, den Request an sich und die Aktualisierung der Resource in der Datenbank zu trennen. Dadurch ist der Controller ausschließlich als Schnittstelle zum Frontend zu sehen, der keine eigentlich Business Logic ausführt, sondern nur das HTTP-Protokoll bedient.

Es soll ein Controller also nie direkt den DbContext referenzieren.

4.1 Service erstellen

Dazu erzeugt man für jeden Controller eine eigene Service-Klasse. Auch hierfür einen eigenen Ordner erstellen, den man sinnigerweise Services nennt.



Dieses Service kommuniziert mit der Datenbank, daher diese über den Dependency Injection Mechanismus im Konstruktor wie gewohnt anfordern.

```
public class CategoriesService
{
    private readonly WebApiDemoContext _db;

    public CategoriesService(WebApiDemoContext db) => _db = db;
}
```

Hinweis: die Datenbank heißt hier WebApiDemoContext, weil das Projekt mit dem eingangs erwähnten Script erstellt wurde. Dabei heißt die DbContext-Klasse dann immer wie das Projekt mit Suffix „Context“.

4.2 Service verwenden

Um das Service verwenden zu können, muss dieses im Service Container registriert werden. Das ist einer der häufigsten Fehler, dass man das vergisst!

Die Lebensdauer der Objekte hängt von AddXXX() ab:

AddTransient()	Objekt wird immer neu erzeugt
AddScoped()	pro Request wird Objekt neu erzeugt
AddSingleton()	Objekt wird nur ein einziges Mal erzeugt, lebt so lange wie der Server

Für unseren Zweck ist es nicht so wesentlich, ob die Instanz einmal oder öfter erzeugt wird:

```
builder.Services.AddDbContext<WebApiDemoContext>(options => options.UseSqlServer(ab:
builder.Services.AddScoped<CategoriesService>();
```

Hinweise:

- Registriert man den Datenbankkontext mit **AddDbContext()** entspricht das einem **AddScoped()**.
- Die Lebensdauer eines Service darf nicht kürzer sein als jenes der Datenbank (ausprobieren Db mit AddDbContext und Service mit AddSingleton)! Selbst überlegen, warum das nicht sein darf/kann!

Damit kann das Service im Konstruktor mit Dependency Injection angefordert werden:

```
[Route("[controller]")]
[ApiController]
public class CategoriesController : ControllerBase
{
    private readonly CategoriesService _categoriesService;
    public CategoriesController(CategoriesService categoriesService)
    {
        _categoriesService = categoriesService;
    }
}
```

Der Controller verwendet jetzt den DbContext nicht mehr direkt.

4.3 Methoden verschieben

Jetzt alle Methoden im Service programmieren, z.B. die GetAll-Methode:

```
public IEnumerable<Category> GetAll()
{
    return _db.Categories.OrderBy(x => x.CategoryName).AsEnumerable();
}
```

Es praktisch nur der Code, der auf den Datenbank-Context zugreift, in das Service zu verschieben:

```
[HttpGet]
public IEnumerable<CategoryDto> Get()
{
    Console.WriteLine("CategoriesController::Get");
    return _categoriesService.GetAll()
        .Select(x => new CategoryDto
        {
            CategoryId = x.CategoryId,
            CategoryName = x.CategoryName,
            Description = x.Description ?? "",
        });
}
```

4.4 DTO Verwendung vereinfachen

Es funktioniert zwar, aber man muss also für jede Instanz eine entsprechende DTO-Instanz erzeugen und die Properties kopieren. Dazu bietet sich an, das mit einer **Extensionmethode** zu lösen (es gibt auch Nuget-Pakete wie z.B. AutoMapper, zur allgemeinen Lösung dieses Problems). Eine derartige Methode ist bereits in ExtensionMethods.cs vorhanden:

```
public static T CopyPropertiesFrom<T>(this T target, object source, string[]? ignoreProperties)
{
    if (target == null) return target;
    if (ignoreProperties == null) ignoreProperties = Array.Empty<string>();
    var propsSource = source.GetType().GetProperties()
        .Where(x => x.CanRead && !ignoreProperties.Contains(x.Name));
    var propsTarget = target.GetType().GetProperties().Where(x => x.CanWrite);

    propsTarget
        .Where(prop => propsSource.Any(x => x.Name == prop.Name))
        .ToList()
        .ForEach(prop =>
        {
            var propSource = propsSource.Where(x => x.Name == prop.Name).First();
            prop.SetValue(target, propSource.GetValue(source));
        });
    return target;
}
```

Damit ist die Controller-Action folgendermaßen zu ändern:

```
[HttpGet]
public IEnumerable<CategoryDto> Get()
{
    Console.WriteLine("CategoriesController::Get");
    return _categoriesService.GetAll()
        .Select(x => new CategoryDto().CopyPropertiesFrom(x));
}
```

4.5 Testen

Mit REST Client testen:

<pre>1 reference @hostname = localhost 1 reference @port = 5000 2 references @url = http://{hostname}:{port} 6 references @categories = {{url}}/categories 0 references @products = {{url}}/products 2 references @categoryId = 5012 ##### Categories Send Request GET {{categories}} HTTP/1.1 ####</pre>	<pre>1 HTTP/1.1 200 OK 2 Connection: close 3 Date: Fri, 15 May 2020 14:38:49 GMT 4 Content-Type: application/json; charset=utf-8 5 Server: Kestrel 6 Transfer-Encoding: chunked 7 8 √ [9 √ { 10 "categoryId": 1, 11 "categoryName": "Beverages", 12 "description": "Soft drinks, coffees, teas, be ers, and ales" 13 }, 14 √ { 15 "categoryId": 2,</pre>
---	---

4.6 Interface pro Service

Es wird empfohlen, neben der Service-Klasse auch ein Interface zu definieren.

```
public interface ICategoriesService
{
    IEnumerable<Category> GetAll();
    Category Delete(int id);
    Category GetSingle(int id);
    Category Insert(Category category);
    Category Update(int id, Category category);
}
```

```
public class CategoriesService : ICategoriesService
{
    private readonly WebApiDemoContext _db;
    public CategoriesService(WebApiDemoContext db) => _db = db;
```

Dieses Interface dann mit der zugehörigen Implementierung in Program.cs beim ServiceContainer registrieren.

```
//builder.Services.AddScoped<CategoriesService>();
builder.Services.AddScoped<ICategoriesService, CategoriesService>();
```

Im Controller jetzt nur dieses Interface anzufordern – man bekommt dann als Instanz die registrierte Implementierung des Interface:

```
public class CategoriesController : ControllerBase
{
    private readonly ICategoriesService categoriesService;
    public CategoriesController(ICategoriesService categoriesService)
    {
        _categoriesService = categoriesService;
    }
}
```

Hinweis: Würde man weiterhin im Controller die Klasse und nicht das Interface anfordern, würde beim Starten eine Exception geworfen – registriert ist nur das Interface!

4.7 Request vs. Reply

Oft bietet sich an, für Requests und Replies (leicht) unterschiedliche DTOs zu verwenden, weil man z.B. bei einem POST keine Id angeben muss bzw. soll.

```
public class CategoryDto
{
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public override string ToString() => $"{CategoryName}";
}

public class CategoryReplyDto : CategoryDto
{
    public int CategoryId { get; set; }
    public override string ToString() => $"{CategoryName} [{CategoryId}]";
}
```

4.8 Post

Analog sieht ein Post aus.

4.8.1 Controller

```
[HttpPost]
public CategoryReplyDto Post([FromBody] CategoryDto categoryDto)
{
    Console.WriteLine($"CategoriesController::Post {categoryDto}");
    var category = _categoriesService.Insert(new Category().CopyPropertiesFrom(categoryDto));
    return new CategoryReplyDto().CopyPropertiesFrom(category);
}
```

4.8.2 Service

```
public Category Insert(Category category)
{
    _db.Categories.Add(category);
    _db.SaveChanges();
    return category;
}
```

Und funktioniert:

```
Send Request
POST {{categories}} HTTP/1.1
Content-Type: application/json

{
  "categoryId": 3,
  "categoryName": "Vegetables",
  "description": "all kinds of vegetables"
}
```

```
HTTP/1.1 200 OK
Connection: close
Date: Mon, 24 May 2021 14:35:44 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{
  "categoryId": 14,
  "categoryName": "Vegetables",
  "description": "all kinds of vegetables"
}
```

Problem: was passiert im Fehlerfall?

4.9 FromQuery/FromBody

Bei den Parametern kann man angeben (muss man aber nicht), woher die Parameter kommen:

- aus dem Body des Requests
- über Query-Parameter

4.9.1 FromQuery

Hier werden die Parameter an die URL mit „?“ als Query-Parameter angehängt. Das ist auch der Default, folgende Methoden sind daher gleichwertig:

```
[HttpGet("[action]")]
public string T1(string first, string last)
{
    return $"T1: {first} {last}";
}
```

```
[HttpGet("[action]")]
public string T2([FromQuery] string first, [FromQuery] string last)
{
    return $"T2: {first} {last}";
}
```

Aufruf:

```
GET {{values}}/t1?first=Hansi&last=Huber
```

4.9.2 FromBody

Die Methodenparameter werden nur im Requestbody gesucht. Im Header muss man dazu den Content-Type mit „application/json“ angeben.

```
[HttpGet("[action]")]
public string T3([FromBody] PersonDto personDto)
{
    return $"T3: {personDto.First} {personDto.Last}";
}
```

```
GET {{values}}/t3 HTTP/1.1
Content-Type: application/json

{
    "first": "Hansi",
    "last": "Huber"
}
```

Es darf nur maximal einen Parameter mit Attribut [FromBody] geben. Daher muss bei mehreren Parametern ein DTO erzeugt werden.

Folgendes funktioniert also nicht:

```
[HttpGet("[action]")]
public string T4([FromBody] string first, [FromBody] string last)
{
    return $"T4: {first} {last}";
}
```

Man bekommt am Backend beim Start eine Exception und am Frontend daher ein „404 Not Found“:

```
GET {{values}}/t4 HTTP/1.1
Content-Type: application/json

{
    "first": "Hansi",
    "last": "Huber"
}
```

```
HTTP/1.1 404 Not Found
Content-Length: 0
Connection: close
Date: Sun, 27 Mar 2022 12:22:52 GMT
Server: Kestrel
```

4.10 ResultFilter

Die Frage ist noch, wo die Transformation von einem Datenbankobjekt in ein DTO erfolgen soll – im Controller oder im Service.

Eine Möglichkeit ist, dies als ResultFilter zu programmieren.

Dabei gibt es eine „normale“ und eine asynchrone Variante, wobei nur die asynchrone zu funktionieren scheint.

4.10.1 Synchron

Die synchrone Variante sähe so aus, funktioniert aber wie erwähnt bei mir nicht (bin dem auch nicht weiter nachgegangen):

```
public class CategoryResultFilterAttribute : ResultFilterAttribute
{
    public override void OnResultExecuted(ResultExecutedContext context)
    {
        var result = context.Result as ObjectResult;
        result.Value = new CategoryReplyDto().CopyPropertiesFrom(result.Value);
    }
}
```

4.10.2 Async

Die asynchrone Variante sieht so aus, sie funktioniert dann auch:

```
public class CategoryResultFilterAttribute : ResultFilterAttribute
{
    public override async Task OnResultExecutionAsync(ResultExecutingContext context,
        ResultExecutionDelegate next)
    {
        var result = context.Result as ObjectResult;
        if (result?.Value == null || result.StatusCode < 200 || result.StatusCode >= 300)
        {
            await next();
            return;
        }
        result.Value = new CategoryReplyDto().CopyPropertiesFrom(result.Value);
        await next();
    }
}
```

4.10.3 Verwenden

Dieser Filter kann dann bei einer Methode als Attribut notiert werden. In diesem Fall wird dann das Objekt nach „Verlassen“ der Action-Methode durch den Filter geschickt, bevor es den Client erreicht.

```
[HttpPost]
[CategoryResultFilter]
public Category Post([FromBody] CategoryDto categoryDto)
{
    Console.WriteLine($"CategoriesController::Post {categoryDto}");
    var category = new Category().CopyPropertiesFrom(categoryDto);
    return _categoriesService.Insert(category);
}
```


5 Background tasks

Manchmal braucht man Services, die nicht in einem Controller benutzt werden, sondern die zu Beginn selbständig starten und Code ausführen sollen.

Man kann das etwas unschön in Program.cs so lösen, wie im letzten Tutorial beschrieben. Zur Erinnerung:

```
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();
```

Speziell dafür vorgesehen sind das Interface `IHostedService` bzw. die Klasse `BackgroundService`, die eben dieses Interface implementiert.

5.1 BackgroundService

Die einfachste Variante implementiert nur die Methode `ExecuteAsync`.

```
public class DummyBackgroundService : BackgroundService
{
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        Console.WriteLine($"DummyBackgroundService::ExecuteAsync");
        //this is running in the main thread
        return Task.Run(() =>
        {
            Console.WriteLine($"DummyBackgroundService: executing Task");
            //this is running asynchronously in the background
        }, stoppingToken);
    }
}
```

5.2 Starten

Ein `BackgroundService` wird ähnlich wie andere Services registriert, jedoch mit der Methode `AddHostedService<>`:

```
builder.Services.AddHostedService<DummyBackgroundService>();
```

5.3 Synchron vs. Asynchron

Alles außerhalb `Task.Run()` blockiert, innerhalb von `Task.Run()` nicht. Bei folgendem Code ist also der Server die ersten 10 Sekunden nicht verfügbar.

```
protected override Task ExecuteAsync(CancellationToken stoppingToken)
{
    Console.WriteLine($"DummyBackgroundService::ExecuteAsync");
    //this is running in the main thread
    Console.WriteLine("Sleeping 10sec...");
    Thread.Sleep(10000);
    Console.WriteLine("Woke up!");
    return Task.Run(() =>
    {
        Console.WriteLine($"DummyBackgroundService: executing Task");
        //this is running asynchronously in the background
        Console.WriteLine("Sleeping 10sec in background...");
        Thread.Sleep(10000);
        Console.WriteLine("Woke up!");
    }, stoppingToken);
}
```

In den ersten 10 Sekunden können daher auch keine Daten geliefert werden.


```
DummyBackgroundService::ExecuteAsync  
Sleeping 10sec...  
Woke up!  
DummyBackgroundService: executing Task  
Sleeping 10sec in background...  
CategoriesController::Get 1  
Woke up!
```

5.4 Datenbank erstellen

Damit kann man jetzt eine Code First Datenbank mit einem BackgroundService erstellen und auch mit Daten befüllen. Damit sichergestellt ist, dass alle Controller bzw. Services die erzeugte Datenbank zur Verfügung haben, muss dieser Code im Synchron-Teil von ExecuteAsync notiert werden.

Es muss noch berücksichtigt werden, dass in ein HostedService kein Scoped Service injected werden kann. Daher muss man sich etwaige Services und somit auch den DbContext über einen ServiceProvider besorgen:

```
private readonly IServiceProvider _serviceProvider;  
public DummyBackgroundService(IServiceProvider serviceProvider) => _serviceProvider = serviceProvider;  
  
protected override Task ExecuteAsync(CancellationToken stoppingToken)  
{  
    Console.WriteLine($"DummyBackgroundService::ExecuteAsync");  
    using IServiceScope scope = _serviceProvider.CreateScope();  
    var db = scope.ServiceProvider.GetRequiredService<DummyDbContext>();  
    db.Database.EnsureDeleted();  
    db.Database.EnsureCreated();  
    Console.WriteLine("Database Ok");  
    int nrPersons = db.Persons.Count();  
    Console.WriteLine($" nrPersons = {nrPersons}");  
    return Task.Run(() =>  
    {  
        Console.WriteLine($"DummyBackgroundService: executing Task");  
    }, stoppingToken);  
}
```

6 StatusCode / ActionResult

Wie bereits besprochen ist der StatusCode eines Requests wesentlich. Die Frage ist daher, wie man diesen StatusCode beeinflussen bzw. setzen kann.

6.1 StatusCode

Man kann jederzeit im Controller den StatusCode selbst setzen, und zwar über `HttpContext.Response.StatusCode`.

```
[HttpGet("{id}")]
public CategoryReplyDto Get(int id)
{
    Console.WriteLine($"CategoriesController::Get {id}");
    if (id < 0)
    {
        HttpContext.Response.StatusCode = 400;
        return null;
    }
    return new CategoryReplyDto().CopyPropertiesFrom(_categoriesService.GetSingle(id));
}
```

6.2 ActionResult<T>

Es gibt auch die Möglichkeit, in der Signatur den Returntyp **ActionResult** anzugeben. Dann kann man mit Methoden wie **Ok()** oder **BadRequest()** den StatusCode bestimmen und muss nicht selbst den Wert angeben.

Damit aber Tools wie Swagger am Returntyp erkennen können, welche Daten tatsächlich zurückgegeben werden, muss immer die generische Variante gewählt werden (siehe dazu auch noch Anmerkungen im Kapitel Swagger).

Also:

- Returntyp ist **ActionResult<T>**
- Im Erfolgsfall wird mit **Ok()** ein ActionResult mit Responsecode 200 erzeugt
- Im Fehlerfall wird mit **BadRequest()** Responsecode 400 „Bad Request“ erzeugt

```
[HttpGet("{id}")]
public ActionResult<CategoryReplyDto> Get(int id)
{
    Console.WriteLine($"CategoriesController::Get {id}");
    if (id < 0) return BadRequest("Negative Ids not allowed!");
    return Ok(new CategoryReplyDto().CopyPropertiesFrom(_categoriesService.GetSingle(id)));
}
```

6.2.1 CreatedAtAction

Üblicherweise ist der Responsecode eines Post 201, wobei auch die URL zurückgegeben wird, unter der man die neu generierte Resource laden kann. Das geht mit **CreatedAtAction**:

```
[HttpPost]
public ActionResult<CategoryReplyDto> Post([FromBody] CategoryDto categoryDto)
{
    Console.WriteLine($"CategoriesController::Post {categoryDto}");
    var category = _categoriesService.Insert(new Category().CopyPropertiesFrom(categoryDto));
    //return Ok(new CategoryReplyDto().CopyPropertiesFrom(category));
    string actionName = nameof(Get);
    var routeValues = new { id = category.CategoryId };
    return CreatedAtAction(
        actionName,
        routeValues,
        new CategoryReplyDto().CopyPropertiesFrom(category)
    );
}
```

Und das bekommt man als Response:

```
HTTP/1.1 201 Created
Connection: close
Content-Type: application/json; charset=utf-8
Date: Sun, 27 Mar 2022 13:14:55 GMT
Server: Kestrel
Location: https://localhost:5001/CategoriesResult/1012
Transfer-Encoding: chunked

{
  "categoryId": 1012,
  "categoryName": "Vegetables",
  "description": "all kinds of vegetables"
}
```

6.2.2 ActionResult

Alle verfügbaren ActionResult findet man unter folgender URL (2.2 ist kein Tippfehler, für 6.x gibt es keine Aktualisierung, Stand 2022-03-27):

<https://docs.microsoft.com/en-us/dotnet/api/system.web.http.apicontroller?view=aspnetcore-2.2#methods>

7 Fehlerbehandlung

Jetzt ist noch ausständig, wie man mit etwaigen Fehlern umgeht. Wie bereits eingangs besprochen sollte Folgendes beachtet werden:

- Der Fehler soll im Header als Responsecode angegeben werden.
- Die Fehlermeldung den Client erreichen
- Im Erfolgsfall die Daten geliefert werden

7.1 Testen

Bei Entity Framework Core darf beim Insert die Id nicht gesetzt sein.

Fügt man also folgende Zeile ein (in CategoriesService.cs), funktioniert es nicht mehr und man kann den Fehlerfall testen:

```
public Category Insert(Category category)
{
    category.CategoryId = 123;
    _db.Categories.Add(category);
    _db.SaveChanges();
    return category;
}
```

7.2 Exception

Exceptions werden automatisch mit Statuscode 500 an den Client geschickt. Dabei ist jedoch die Fehlermeldung sehr umfangreich.

```
HTTP/1.1 500 Internal Server Error
Connection: close
Content-Type: text/plain; charset=utf-8
Date: Sun, 27 Mar 2022 12:46:38 GMT
Server: Kestrel
Transfer-Encoding: chunked

Microsoft.EntityFrameworkCore.DbUpdateException: An error occurred while saving the entity changes. See the inner exception for details.
---> Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity column in table 'Categories' when IDENTITY_INSERT is set to OFF.
```

7.3 Globaler Exceptionhandler

Exceptions können auch durch Angabe einer zusätzlichen Middleware an zentraler Stelle gecatcht werden, also in Startup.cs in der Methode Configure (wichtig: vor app.UseEndpoints()):

```
app.UseAuthorization();

app.UseExceptionHandler(config =>
{
    config.Run(async context =>
    {
        context.Response.StatusCode = 500;
        context.Response.ContentType = "application/json";
        var error = context.Features.Get<ExceptionHandlerFeature>();
        if (error != null)
        {
            await context.Response.WriteAsync(
                $"Exception: {error.Error?.Message} {error.Error?.InnerException?.Message}");
        }
    });
});

app.MapControllers();
```

Man braucht dabei folgende **using**:

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Http;
```

Damit sieht die Fehlermeldung so aus:

```
HTTP/1.1 500 Internal Server Error
Connection: close
Content-Type: application/json
Date: Sun, 27 Mar 2022 12:44:12 GMT
Server: Kestrel
Cache-Control: no-cache,no-store
Expires: -1
Pragma: no-cache
Transfer-Encoding: chunked
```

```
Exception: An error occurred while saving the entity changes. See the inner exception for details. Cannot
insert explicit value for identity column in table 'Categories' when IDENTITY_INSERT is set to OFF.
```

7.4 Model Validation

Neben den oben besprochenen Fehlerbehandlungen kann man auch noch die übertragenen Daten durch Annotationen im DTO validieren lassen.

So sollen z.B. der Name und die Beschreibung eine bestimmte Länge haben müssen:

```
public class CategoryDto
{
    [MinLength(3)]
    public string CategoryName { get; set; }
    [MinLength(10)]
    public string Description { get; set; }
}
```

Diese Daten werden bei WebApi automatisch überprüft und diese Information mittels ModelState IsValid zur Verfügung gestellt.

Im Response werden dann entsprechende Fehlermeldungen generiert:

```
POST {{categories}} HTTP/1.1
Content-Type: application/json

{
    "categoryName": "v",
    "description": "all"
}
```

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Mon, 24 May 2021 14:57:44 GMT
Content-Type: application/problem+json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "00-4981c21c75d64c4bb6e295540499e88b-5c381b40a9bada41-00",
  "errors": {
    "Description": [
      "The field Description must be a string or array type with a minimum length of '10'."
    ],
    "CategoryName": [
      "The field CategoryName must be a string or array type with a minimum length of '3'."
    ]
  }
}
```

8 Zusammenfassung

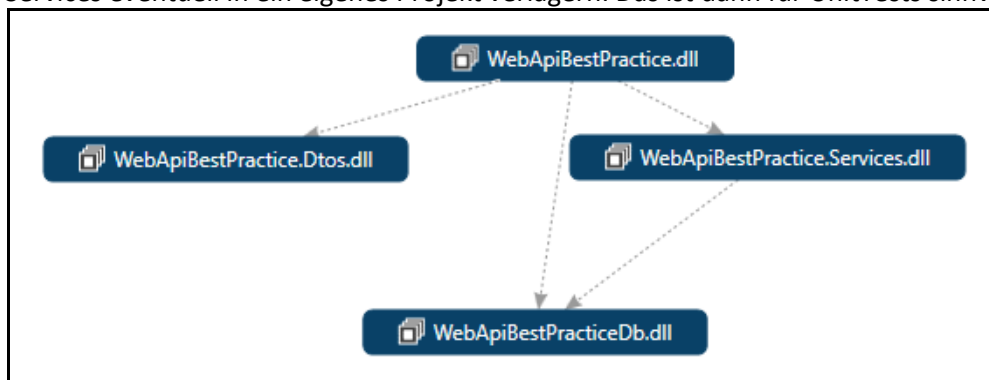
Die Frage stellt sich nun, ob man bei jedem Projekt alle oben gezeigten Möglichkeiten nutzen muss. Mein Vorschlag dazu ist:

Absolutes Muss bei jedem Projekt sind:

- DTOs
- Service
- eigene Reply-/Request-DTO bei Bedarf - ergibt sich aber automatisch aus der Aufgabenstellung
- Globaler Exception-Handler
- BackgroundService bei Code First Datenbanken

Optional:

- Service-Interface: bei kleineren Projekten (wie Schulübung) nicht, man kann bei Bedarf mit Refactoring ziemlich schnell die Verwendung eines Interfaces nachträglich hinzufügen. Bei einer Diplomarbeit ist es meiner Meinung nach aber ein Muss.
- ResultFilter: sind einerseits schnell programmiert und in der Verwendung einfach.
- Services eventuell in ein eigenes Projekt verlagern. Das ist dann für UnitTests sinnvoller.



9 Unit Test

Ein Vorteil von Services ist, dass man diese sehr einfach testen kann, ohne mit irgendwelchen Kommunikationsstrukturen mit dem Frontend in Berührung zu kommen.

Dazu vorgehen wie im entsprechenden Tutorial besprochen ein xUnit Projekt erstellen.

9.1 UnitTest-Klasse

Die Tests selbst wieder in einer ganz normalen Klasse implementieren. Üblicherweise verwendet man dazu nicht die Datenbank des laufenden Projekts, sondern eine, die nur die Stamm- bzw. Testdaten enthält.

Um sicherzustellen, dass auch immer wirklich mit demselben Stand gestartet wird, könnte man die Datenbank vorher kopieren.

Das soll hier auch so gemacht werden, und außerdem zum Testen eine Sqlite-Datenbank verwendet werden:

```
public class BackendTests
{
    private readonly NorthwindContext db;
    private readonly ICategoryService categoryService;
    public BackendTests()
    {
        string testDbSource = @"D:\Temp\Northwnd.sqlite";
        string testDb = @"D:\Temp\NorthwndTest.sqlite";
        Console.WriteLine($"copying {testDbSource} --> {testDb}");
        File.Copy(testDbSource, testDb, overwrite: true);
        var options = new DbContextOptionsBuilder<NorthwindContext>()
            .UseSqlite($"data source={testDb}")
            .Options;
        db = new NorthwindContext(options);
        categoryService = new CategoryService(db);
    }
}
```

Hier sieht man auch ganz deutlich den Vorteil von Dependency Injection: Das Service erzeugt nicht selbst eine Instanz für den Zugriff auf die Datenbank, sondern nimmt jene, die über den Konstruktor zur Verfügung gestellt wird („Inversion of Control“).

Wenn man wie oben vorgeschlagen die Services in ein eigenes Projekt gibt, erkennt man, dass man keinen Verweis auf das eigentliche Backend braucht.

9.2 Testmethoden

Die Tests programmiert man dann wie gewohnt:

```
[Fact]
public void T01_NumberCategories()
{
    Console.WriteLine("T01_NumberCategories");
    int actual = categoryService.GetAll().Count();
    int expected = 8;
    actual.Should().Be(expected, "because initially the Northwind database has 8 categories");
}
```

Oder:

```
[Fact]
public void T02_AddCategory()
{
    Console.WriteLine("T02_AddCategory");
    categoryService.Insert(new Category
    {
        CategoryId = 9,
        CategoryName = "TestCategory",
        Description = "Test description"
    });
    int actual = categoryService.GetAll().Count();
    int expected = 9;
    actual.Should().Be(expected, "because inserting categories should be successful");
}
```

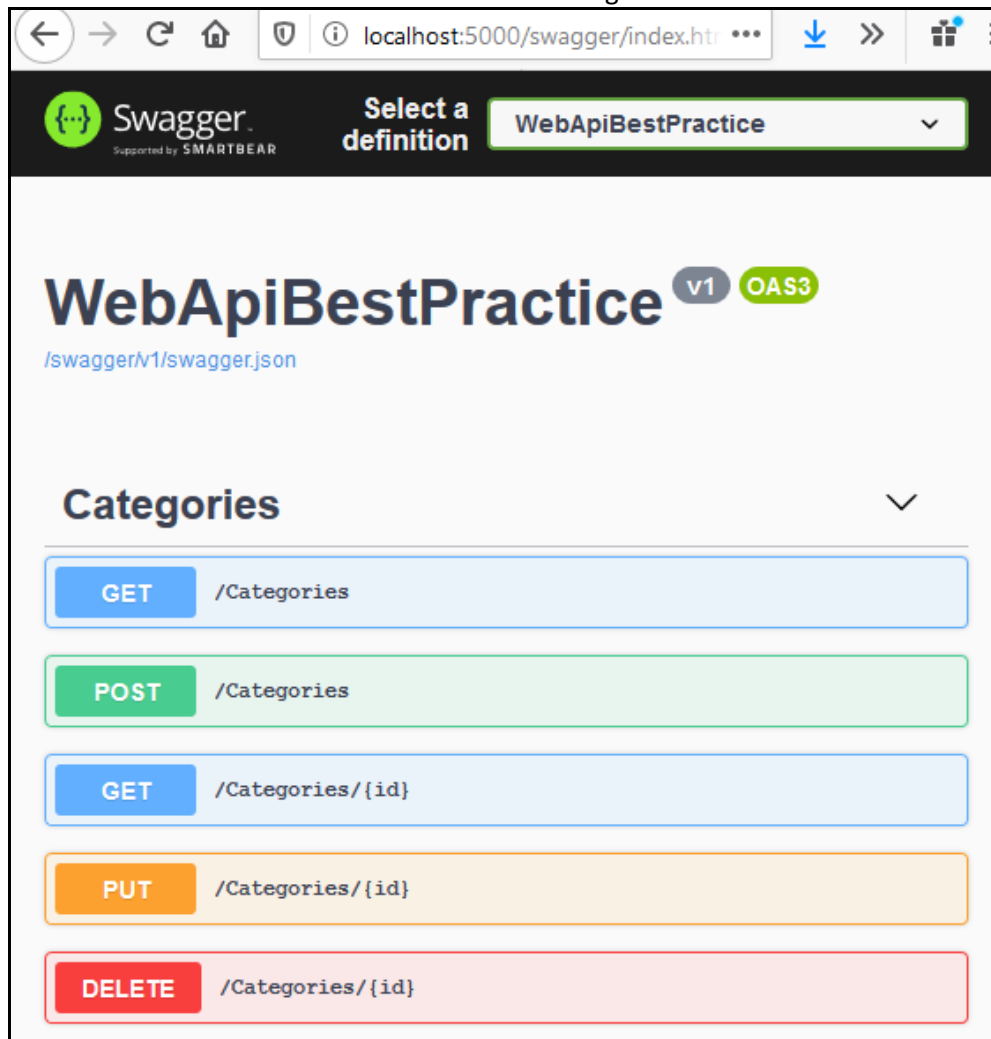
Durch Kopieren der Datenbank ist sichergestellt, dass zu Beginn immer 8 Einträge in der Tabelle sind.
Zur Erinnerung: Vor jeder Test-Methode im Unittest wird der Code im Konstruktor ausgeführt!

10 Swagger/OpenAPI

Swagger bzw. OpenAPI ermöglicht es, Informationen aus einem WebAPI auszulesen. Diese Information wird in einer JSON-Datei zur Verfügung gestellt. Damit kann man dann Zugriffscode für unterschiedliche Programmiersprachen erzeugen.

10.1 API checken

Mit dem Browser muss man dann auf die URL <http://localhost:5000/swagger> navigieren. Dort werden dann auch alle Methoden des Backends mit ihren Routen aufgelistet:



Damit kann man jetzt alle Routen des Backends testen. Diese Routen werden von Swagger automatisch gefunden.

10.1.1 Route testen

Die einzelnen Routen kann man aufklappen und testen, indem man zuerst auf „Try it out“ und dann auf den „Execute“-Button klickt:

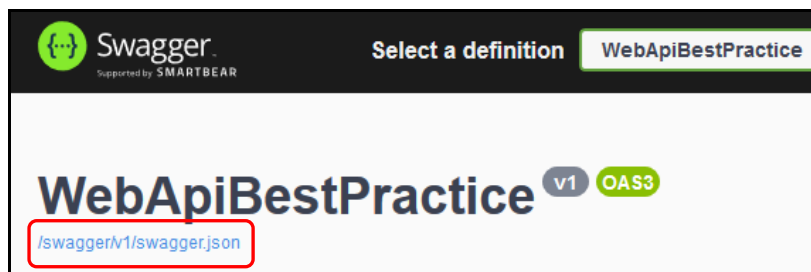
GET /Categories/{id}
Parameters
Try it out
Name
Description
id * required
integer
(path)
Responses
Code
Description
Links
200
Success
No links

GET /Categories/{id}
Parameters
Cancel
Name
Description
id * required
integer
(path)
1
Execute
Responses
Code
Description
Links
200
Success
No links

Responses
Curl
curl -X GET "http://localhost:5000/Categories/1" -H "accept: */*"
Request URL
http://localhost:5000/Categories/1
Server response
Code
Details
200
Response body
{
"categoryId": 1,
"categoryName": "Beverages",
"description": "Soft drinks, coffees, teas, beers, and ales"
}
Download
Response headers
content-type: application/json; charset=utf-8
date: Tue, 29 Oct 2019 10:39:48 GMT
server: Kestrel
transfer-encoding: chunked
Responses
Code
Description
Links
200
Success
No links

10.1.2 swagger.json

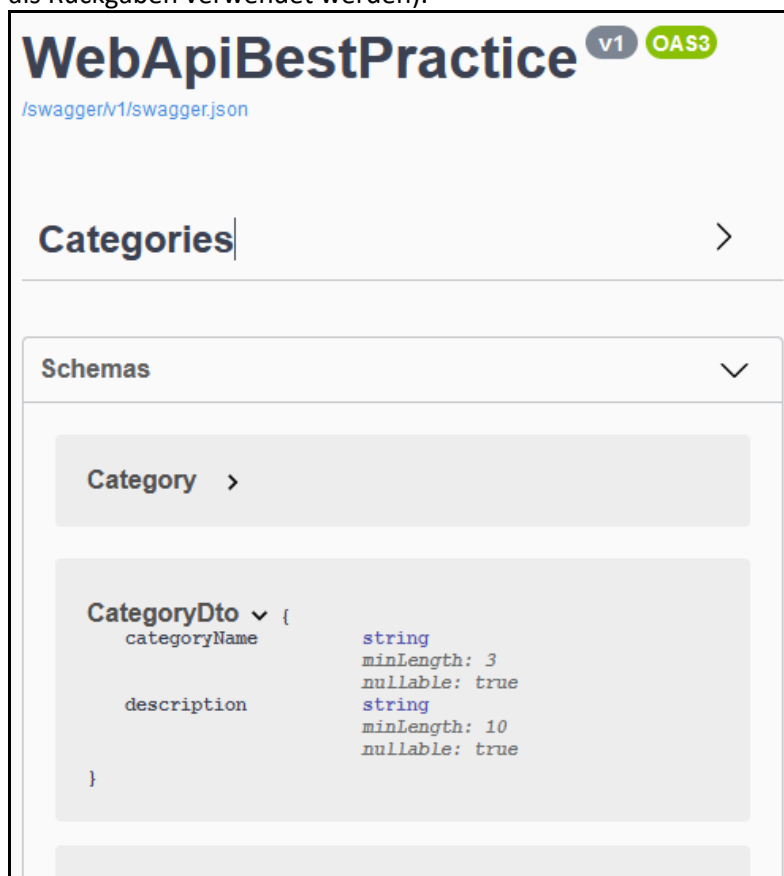
Die gesamte API-Information ist über die Datei **swagger.json** zugänglich. Diese Datei existiert nicht als Datei sondern ist nur über den Link erreichbar.



JSON	Rohdaten	Kopfzeilen
Speichern	Kopieren	Alle einklappen Alle ausklappen
<pre> openapi: "3.0.1" info: {} paths: /Categories: {} /Categories/{id}: get: tags: 0: "Categories" parameters: 0: name: "id" in: "path" required: true schema: type: "integer" format: "int32" responses: 200: description: "Success" put: {} delete: {} /Values/GetCategories: {} components: {} </pre>		

10.1.3 Schemas

Bei den Schemas werden alle potenziellen Rückgabetypen aufgelistet (das heißt nicht, dass diese auch tatsächlich als Rückgaben verwendet werden).



10.2 ActionResult<T>

Swagger analysiert ja die verschiedenen Routen und eruiert daraus die Parameter bzw. den Returntype. Bei den Methoden ist aber der Returntype **ActionResult** und somit nicht erkennbar, welcher Typ tatsächlich zurückgegeben wird.

Daher sollte man bei Verwendung von Swagger anstelle von **ActionResult** den generischen Typ **ActionResult<T>** retournieren, wobei T den tatsächlichen Typ des Responses angibt.

Zum Vergleich:

Mit **ActionResult**:

```
[HttpGet("{id}")]
public IActionResult Get(int id)
```

```

  /Categories/{id}:
    get:
      tags:
        - Categories
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
            format: int32
      responses:
        200:
          description: Success
    put: {}

```

Mit **ActionResult<CategoryReplyDto>**:

```
[HttpGet("{id}")]
public ActionResult<CategoryReplyDto> Get(int id)
```

Eine andere Möglichkeit wäre die Notation mit `ProducesResponseType()`:

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(CategoryReplyDto), (int)HttpStatusCode.OK)]
[CategoryResultFilter]
public IActionResult Get(int id)
```

In beiden Fällen wird der Typ richtig erkannt:

```

  /Categories/{id}:
    get:
      tags: [-]
      parameters: [-]
      responses:
        200:
          description: Success
          content:
            text/plain: {}
            application/json:
              schema:
                $ref: "#/components/schemas/CategoryReplyDto"
            text/json: {}

```