

.NET 5 (Micro-) service erstellen

i Der folgende Artikel beschreibt, wie ein neuer Microservice implementiert und in eine bestehende Microservice Architektur integriert werden kann. Entsprechend kann auch ein Monolith erstellt werden, wobei in diesem Fall die Hinweise zur Anbindung an eine Message Queue bzw. an den OIDC Provider ignoriert werden können.

Technologie Stack

Die Anleitung basiert auf den nachfolgenden Sprachen, Technologien und Frameworks:

- SQL Server und relationale Datenbanken
- Entity Framework Core (Code first, code migrations)
- Linq
- .NET 5.0
- C#
- Rabbit MQ

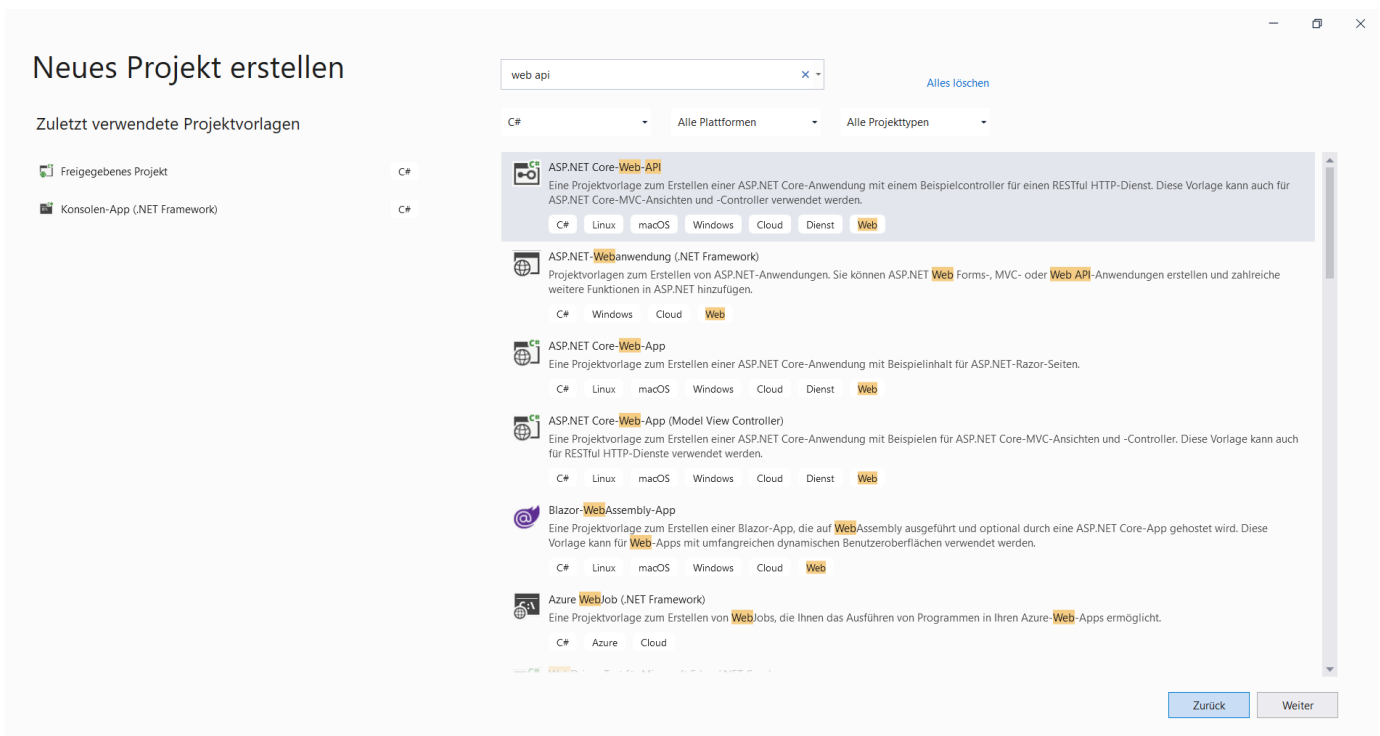
Darüber hinaus werden für die Implementierung die folgenden Konzepte verwendet:

- SOLID principles
- Onion Architektur
- Dependency injection
- Objektorientierte Programmierung

Projekt erstellen

i Im ersten Schritt wird ein neues Web Projekt mit Visual Studio erstellt. Da wir mit einem Web-Client über HTTP kommunizieren möchten, nutzen wir das .NET Core Web-API Template (vgl. Abbildung 1).

w Nicht jeder Microservice braucht unbedingt eine Web API! Wenn die Anforderung lediglich einen Hintergrund-Service vorsieht, welcher mit der Message Queue kommuniziert, kann auch ein .NET Background Service verwendet werden.



Nachdem ein Name vergeben wurde, werden die Templates für die verschiedenen Layer erzeugt (vgl. Onion Architecture). Wie in Abbildung 2 zu erkennen, enthält das Template bereits einen Ordner Controller, in welchem bereits der `WeatherForecastController.cs` erzeugt wurde. Künftige Controller (Konstrukte, welche die RESTful Schnittstelle zu den Clients definieren) sollten in diesem Ordner hinzugefügt werden.

Der Controller und die Klasse `WeatherForeCast[Controller].cs` werden nicht gebraucht. Entweder löschen oder ignorieren.

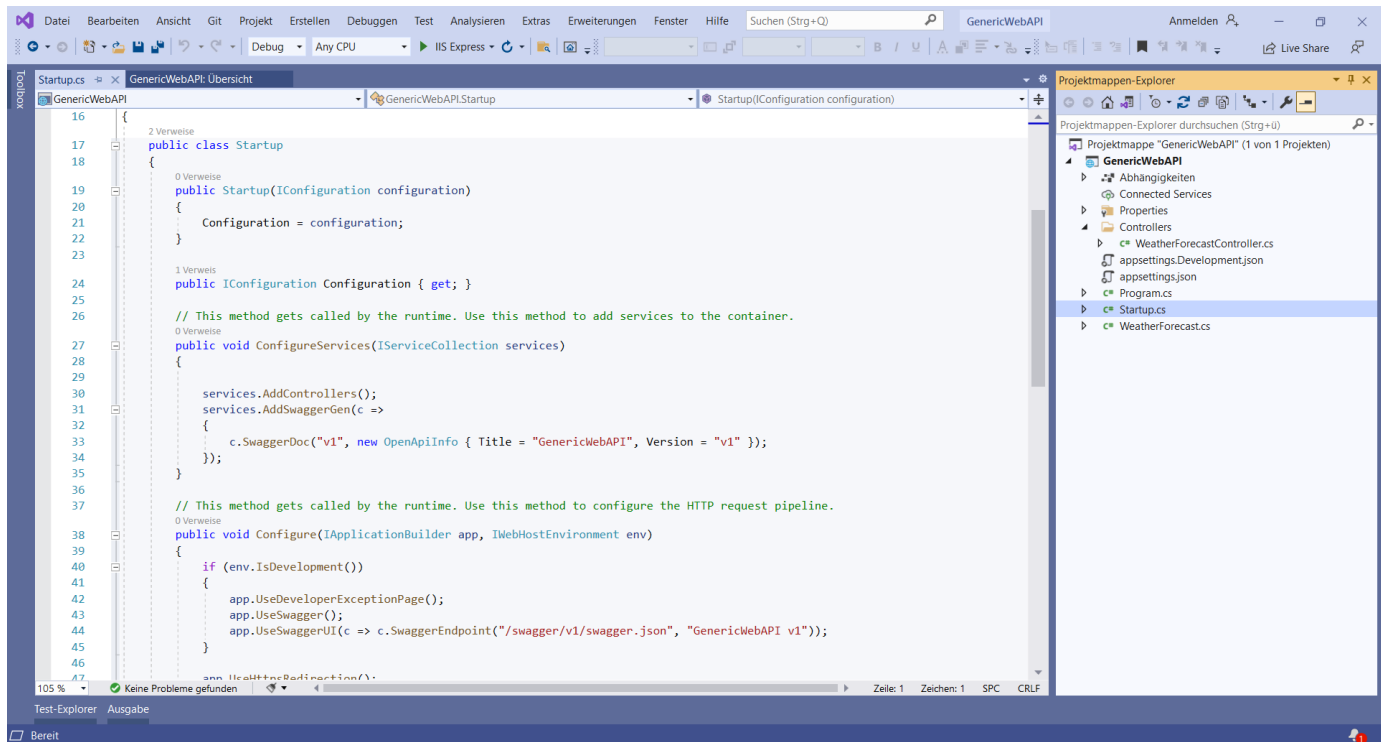
Darüber hinaus wurden die Klassen `Program.cs` und `Startup.cs` generiert. Die Klasse `Program.cs` stellt den Einstiegspunkt für das Programm bereit. Die Klasse `Startup` definiert die Methoden `ConfigureServices` und `Configure`. Beide Methoden werden beim Starten des Service durch die Runtime aufgerufen.

ConfigureServices

Diese Methode definiert die Services, welche den Controllern mittels Dependency Injection (DI) bereitgestellt werden. Das Template definiert dabei bereits 2 Aufrufe: zum einen werden die Controller initialisiert (`services.AddControllers()`) und zum anderen wird die OpenAPI Swagger Dokumentation registriert. Diese wird im späteren Verlauf die Endpunktdefinition bereitstellen.

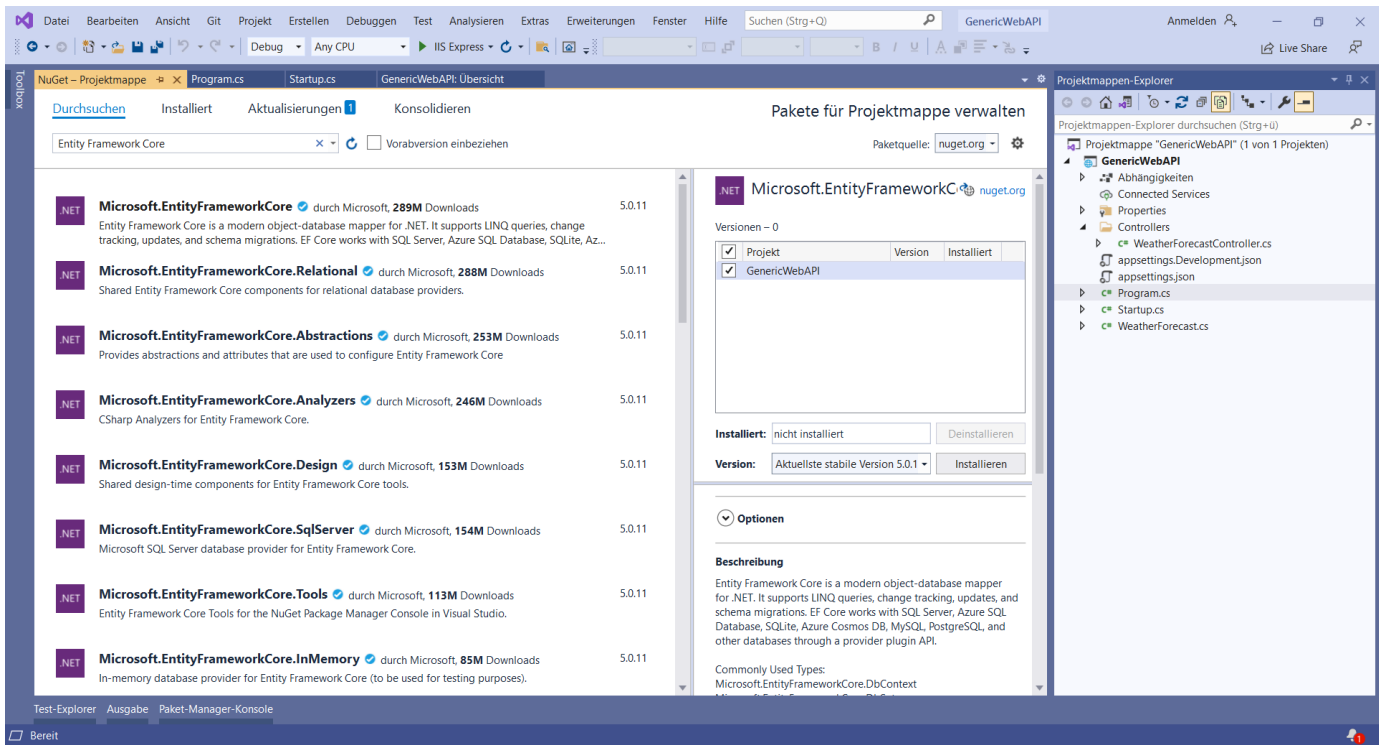
Configure

Diese Methode konfiguriert die Middleware des Services. Unter anderem muss an dieser Stelle die Seed Methode der Datenbank aufgerufen werden.



Datenbank anlegen

Um Daten bereitzustellen und neue Daten zu speichern, muss zunächst ein Datenmodell vorgesehen werden. Dazu sollte Entity Framework Core genutzt werden. Das Framework stellt einen Automatisierten Data-Mapper bereit, um relationale Datenbanken Code-First (oder auch Database-First) zu generieren (an den Service anzubinden). Frameworks werden im Visual Studio mittels NuGet bezogen (Extras NuGet Paket Manager NuGet Pakete verwalten ...). Im NuGet Paket Manager kann anschließend nach Entity Framework Core gesucht werden (vgl. Abbildung 3).



Um Entity Framework mit SQL Server zu nutzen, muss zusätzlich das Paket `Microsoft.EntityFrameworkCore.SqlServer` installiert werden. Es gibt ebenfalls Konnektoren für Postgres, MySQL, Sqlite usw.

i Damit die Migrationen ausgeführt werden können ist es zusätzlich wichtig, das Paket `Microsoft.EntityFrameworkCore.Tools` zu installieren.

Also müssen für dieses Tutorial die folgenden Pakete installiert werden:

- `Microsoft.EntityFrameworkCore`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`

Datenbankkontext generieren

Um mit die Interaktion zwischen Datenbank und Service zu initiieren werden Datenbank Kontexte verwendet. Einen solchen erzeugen wir im nächsten Schritt (vgl. Abbildung 5). Unsere Datenbank wird es ermöglichen, Autos und zugehörige Entitäten zu verwalten. Die notwendigen Entitäten sind im Folgenden beschrieben (vgl. Abbildung 4).

`Car.cs`

Attribut	Typ	Semantik
Guid	Guid	Identifizier
Created	DateTime	Erstelldatum
Series	string	Serie (z.B. 3er)
Type	CarType	Typ (z.B. Cabrio)
Start	DateTime	Start Bau
End	Nullable DateTime	Ende Bau
Power	int	Leistung in kW
ProducerGuid	Nullable Guid	Hersteller Identifizier
Producer	CarProducer	Hersteller

CarProducer.cs

Attribut	Typ	Semantik
Guid	Guid	Identifizier
Created	DateTime	Erstelldatum
Name	string	Herstellernamen (z.B. BMW)
Founded	DateTime	Gründungsdatum
Cars	ICollection Cars	Set aus Fahrzeugen

The screenshot shows the Visual Studio IDE with the 'GenericWebAPI' project open. The 'CarProducer.cs' file is selected in the 'Projektmappe-Explorer' on the right. The code in the editor shows the 'CarProducer' class, which is a partial class in the 'GenericWebAPI.Models' namespace. It has a 'Guid' property, a 'string' property 'Name', a 'DateTime' property 'Founded', and a 'ICollection<Car>' property 'Cars'. The 'Cars' property is initialized with a new 'HashSet<Car>'.

```

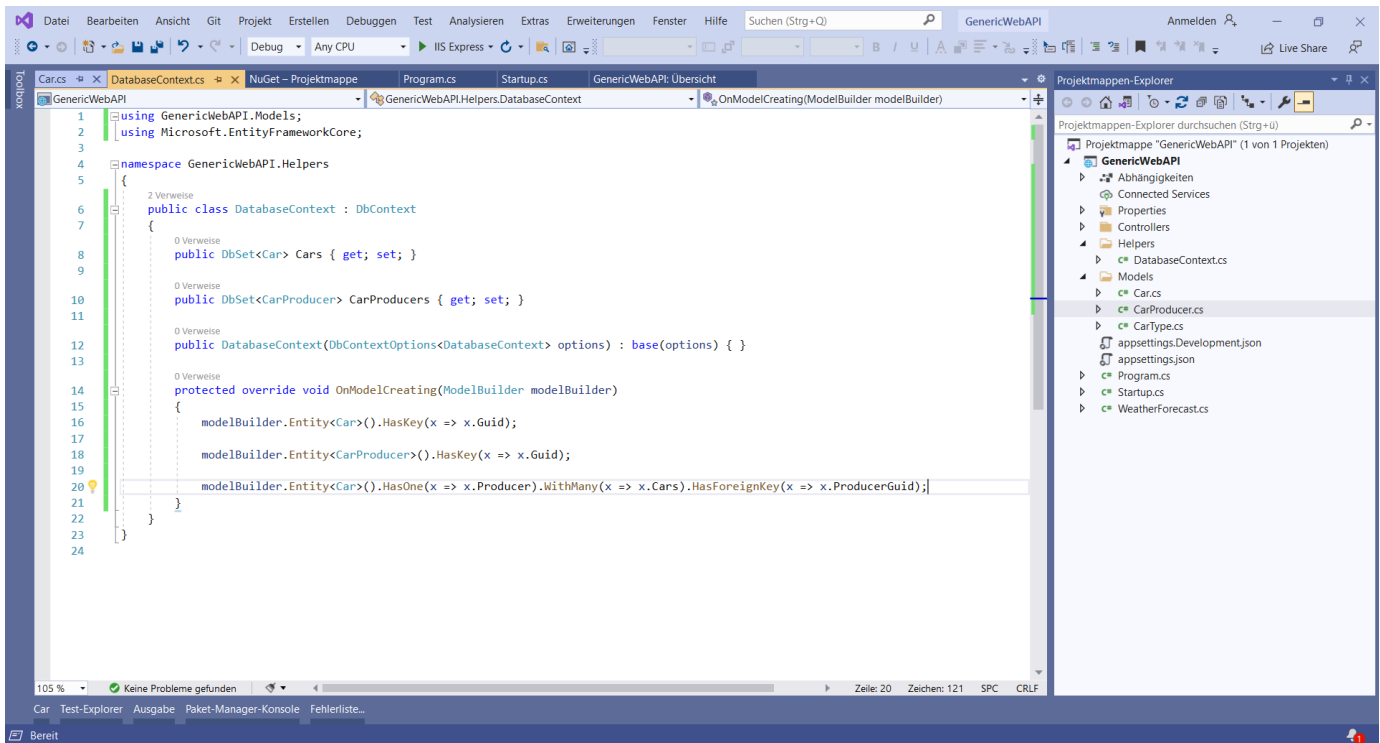
1 using System;
2 using System.Collections.Generic;
3
4 namespace GenericWebAPI.Models
5 {
6     4 Verweise
7     public partial class CarProducer
8     {
9         1 Verweis
10        public Guid Guid { get; set; }
11
12        0 Verweise
13        public string Name { get; set; }
14
15        0 Verweise
16        public DateTime Founded { get; set; }
17
18        4 Verweise
19        public partial class CarProducer
20        {
21            1 Verweis
22            public ICollection<Car> Cars { get; set; } = new HashSet<Car>();
23        }
24    }
25 }

```

Warum werden partial Klassen verwendet?

Microservice Architekturen sollten auf einem geteilten Domainmodell basieren (z.B. ein MS Bond Modell). Es hat sich gezeigt, dass sich Modelle am besten pflegen lassen, wenn die Modelle dynamisch in die Services kompiliert werden können und die servicespezifischen Extensions unberührt in partial classes ausgelagert sind.

Servicespezifisch ist in diesem Fall die 1:n Relationen zwischen Autos und Herstellern.



Wie in Abbildung 5 zu sehen werden im Datenbankkontext alle in der Datenbank vorgesehenen Entitäten zusammen mit den existierenden Fremdschlüsselbeziehungen definiert. Demnach existieren in der Datenbank sowohl Autos und Hersteller und eine 1:n Relation zwischen diesen (ein Auto hat maximal einen Hersteller, ein Hersteller n Autos). Der Fremdschlüssel ist im Schlüssel `ProducerGuid` definiert und verweist auf einen Hersteller.

i Übrigens: im Konstruktor des Datenbank Kontexts ist erstmalig in der App DI zu beobachten. So empfängt der Kontext die Datenbank Optionen, welche im folgenden definiert werden.

Datenbankoptionen

Damit der Service weiß, mit welcher Datenbank er sich verbinden soll, ist es notwendig, einen entsprechenden Connection String für die Anbindung vorzusehen. Dazu muss ein Connection String in der Konfigurationsdatei `appsettings.json` angegeben werden.

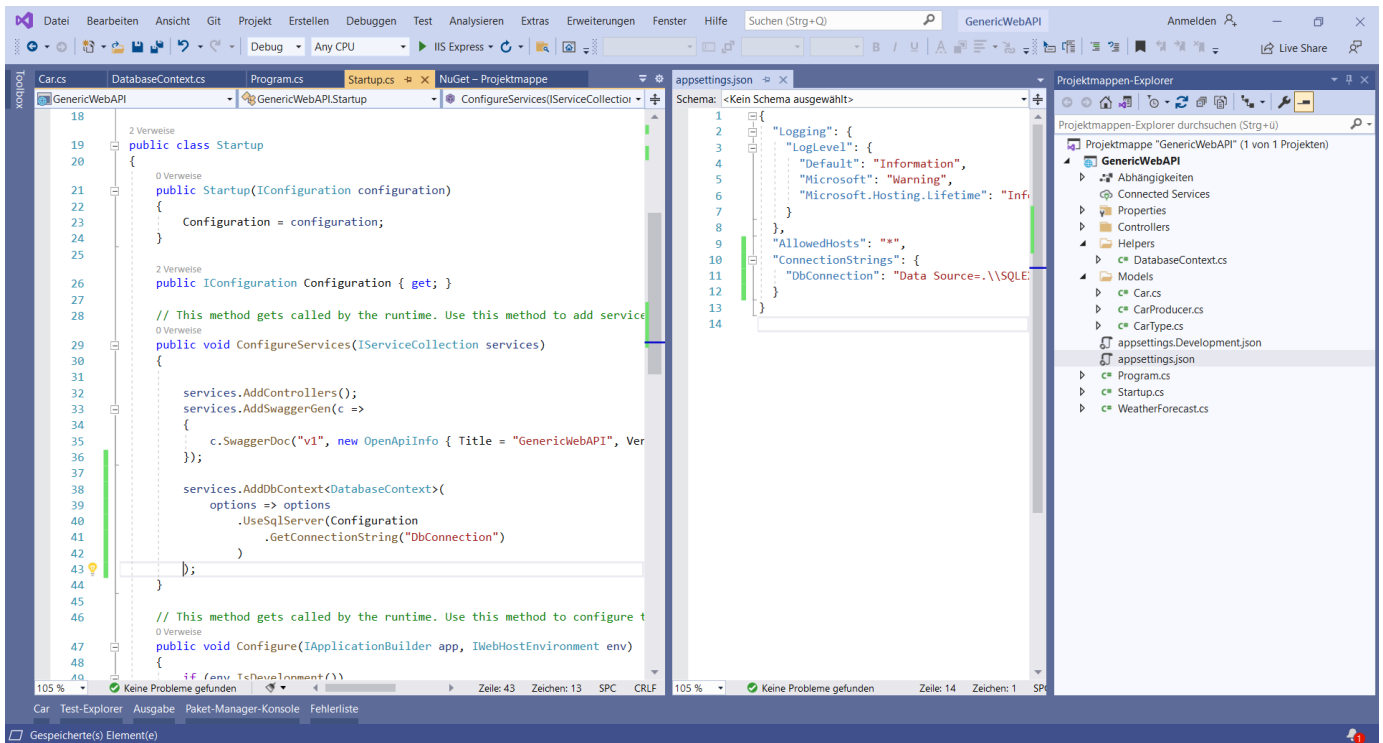
Beispiel:

```

Data Source=.\SQLEXPRESS;Initial Catalog=CarServiceData;Trusted_Connection=True;
MultipleActiveResultSets=true;

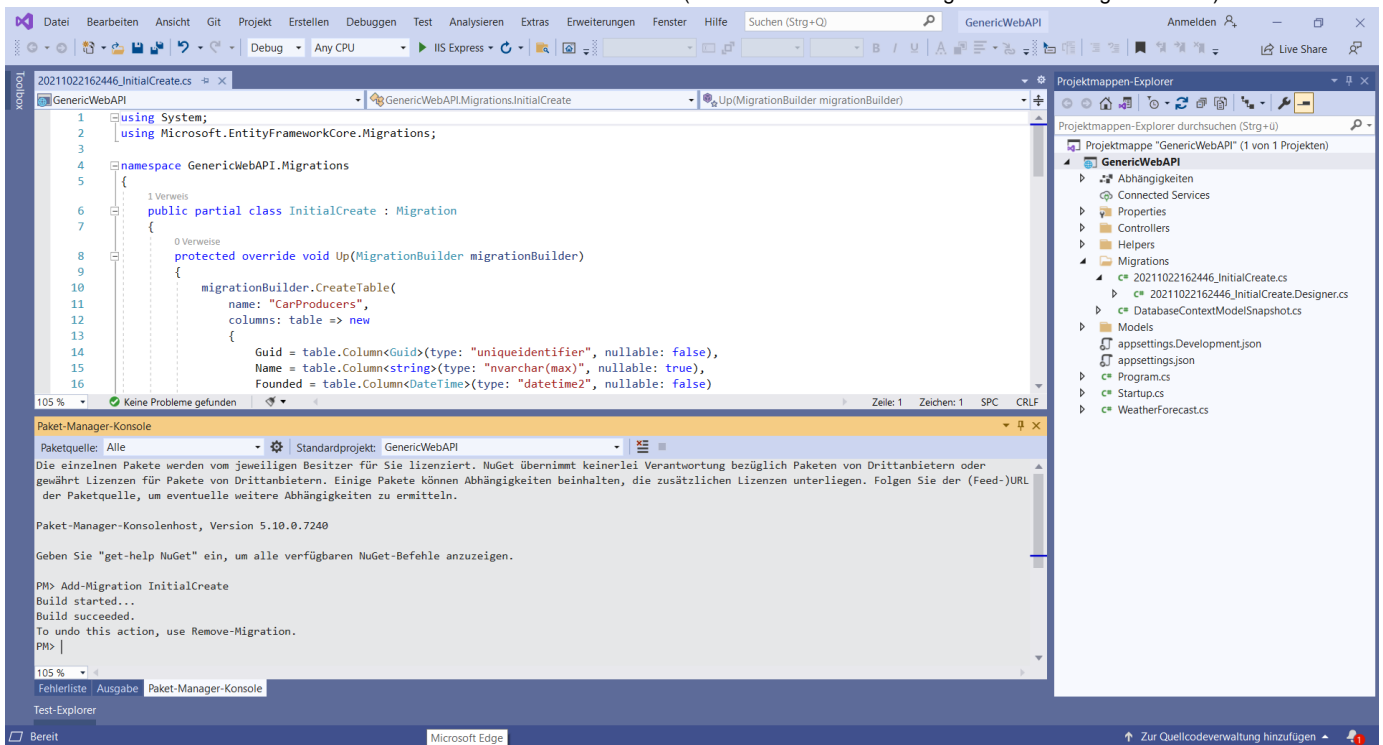
```

Diesen nutzen wir in der `Startup.cs` und registrieren bei der DI eine Datenbankkonfiguration für den Kontext vom Typ `DatabaseContext`. Diese Optionen greifen auf den Connection String mit der Bezeichnung `DbConnection` aus der Applikationskonfiguration zu (vgl. Abbildung 6)



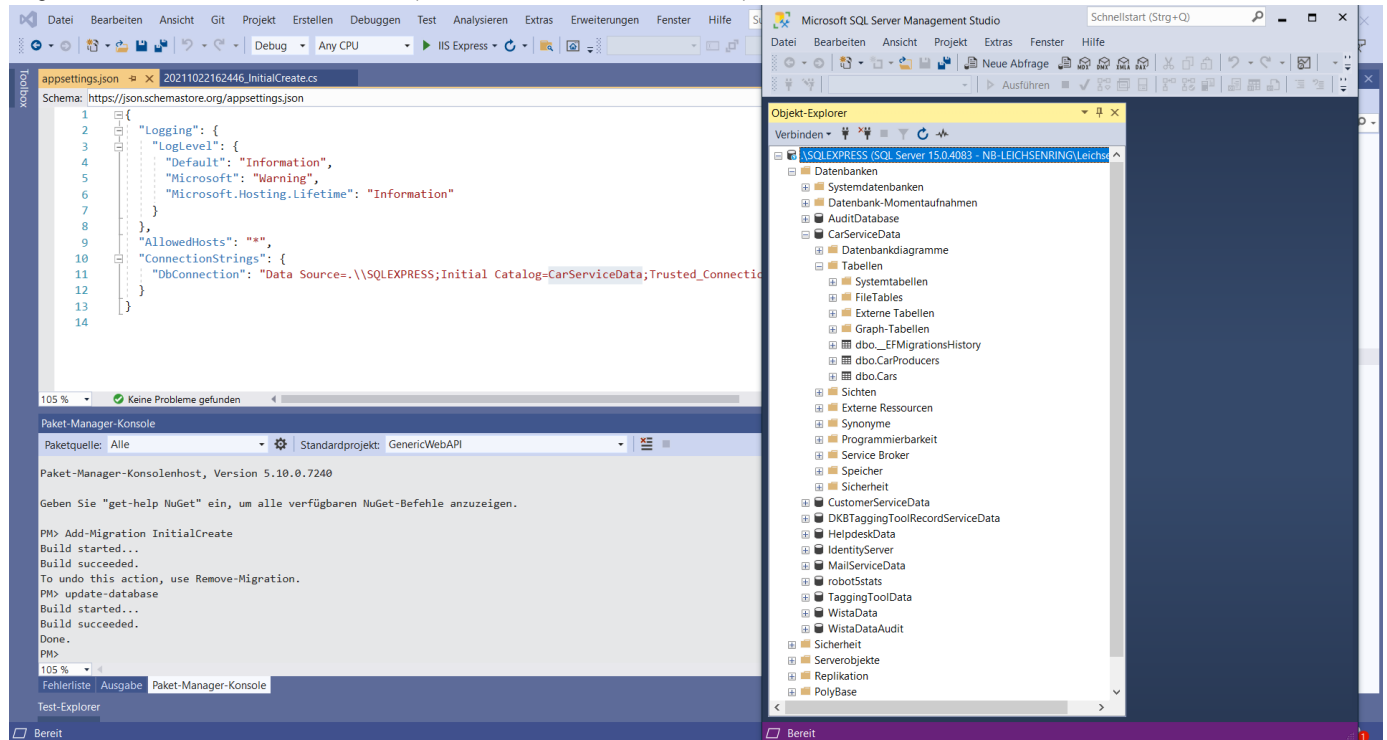
Datenbank Migration erstellen

Änderungen am Datenbankkontext werden mit Hilfe von Migrationen in die Datenbank überführt. Dies erfolgt mit den EntityFrameworkCore.Tools. Diese lassen sich am besten über die NuGet Konsole bedienen (Extras NuGet Paket Manager Paket Manager Konsole).



Um eine Migration zu erzeugen sollte der Befehl `Add-Migration [MIGRATIONNAME]` genutzt werden (für `MIGRATIONNAME` kann ein selbstgewählter Name vergeben werden, ich nutze für die erste Migration immer die Bezeichnung `InitialCreate`). Da nur die Klasse `DatabaseContext` von `DbContext` erbt muss kein Parameter `-Context` übergeben werden. Als Resultat wird der Ordner `Migrations` erzeugt (vgl. Abbildung 7).

Um nun aus dieser Migration ein relationales Schema zu erzeugen, muss der Befehl `Update-Database` ausgeführt werden (vgl. Abbildung 8). Wenn auch dieser Befehl erfolgreich durchgeführt wird, dann sollte in unserem DBMS die Datenbank mit dem im `ConnectionString` vorgesehenen Namen erstellt worden sein (in meinem Fall `CarServiceData`).



i Sie möchten das Datenbankschema ändern? Vergessen Sie nicht, eine neue Migration zu erzeugen und die Datenbank erneut zu migrieren. Erst danach stehen die neuen Felder, Tabellen und FKs bereit.

Repository und Service erzeugen + DI Registrierung

Anschließend müssen Repository und Service für die Manipulation der Entität `Car` erzeugt werden. Dieser Entwurf basiert auf der Onion Architektur. Entscheidend ist dabei, dass wir Methoden für alle CRUD Operationen bereitstellen.

Zunächst das Repository: es stellt einen Wrapper für die Datenbank dar. Dadurch, dass die Methoden nicht direkt im Service definiert werden, ist es möglich, die Methoden in dritten Services wiederzuverwenden. Wenn wir beispielsweise ein `Car` im `CarProducer` Service laden möchten, dann kann einfach eine neue Instanz des Repos instanziiert werden.

i Was ist ein ResponseWrapper?

Da wir dem Client Details zu seiner Anfrage zurückgeben möchten, nutzen wir ein custom Class namens `ResponseWrapper`. Diese ermöglicht es, den HTTP Status und den Abfrage-Content zu wrappen und zum Controller weiterzuleiten.

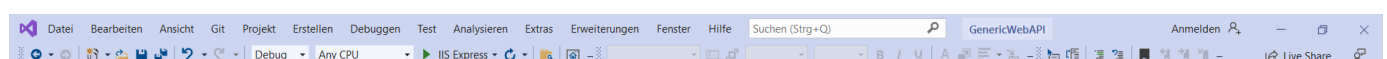
i Was ist eine PaginationPage?

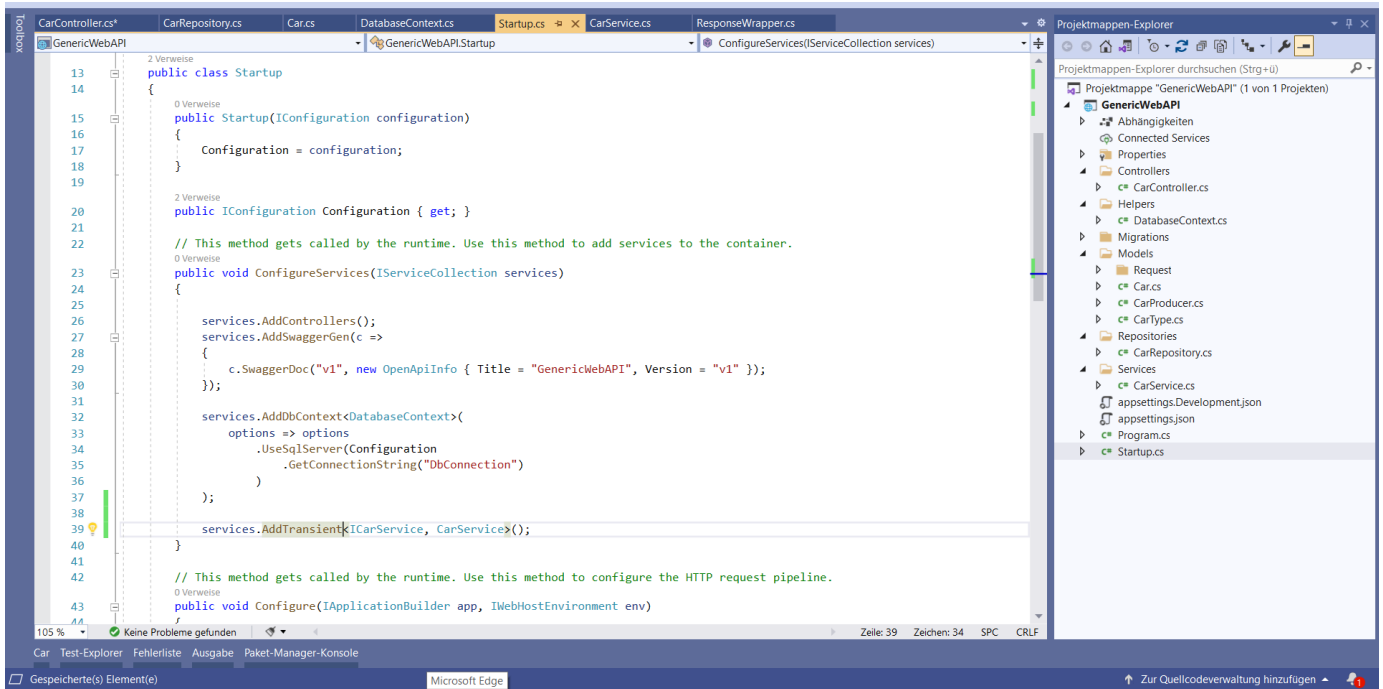
Die `PaginationPage` dient dazu, Datenausschnitte zu definieren. Man stelle sich vor, das wir später 2000 Autos in der Datenbank haben. Da wir beim Start der Homepage nicht alle 2000 brauchen, sondern vielleicht nur die 20 Beliebtesten, nutzen wir `Pagination`. Demnach enthält die Page die Ausschnittsdefinition, also z.B. Seite 1 bestehend aus 20 Einträgen.

Ausschlaggebend dabei ist natürlich die Sortierung, welche die Ausschnittsdefinition bestimmt. In unserem Beispiel sortieren wir einfach nach dem Erstelldatum.

Um den `CarService` mittels DI bereitzustellen, muss dieser in der `Startup.cs` registriert werden. Da wir diesen Service dynamisch bei Bedarf bereitstellen wollen, registrieren wir den Service als `Transient`.

i Der Unterschied wird sehr gut auf StackOverflow erklärt: <https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences>





CarRespository



Achtung!

Inline-Dokumentation ist entscheidend. Undokumentierter Code - egal wie ausgereift - ist nicht fertig!

```

using GenericWebAPI.Helpers;
using GenericWebAPI.Models;
using GenericWebAPI.Models.Request;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace GenericWebAPI.Repositories
{
    public class CarRepository
    {
        /// <summary>
        /// the method returns all cars in accordance to the passed
        pagination page
        /// </summary>
        /// <param name="paginationPage">pagination page</param>
        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>set of cars</returns>
        public async Task<ICollection<Car>> GetCarsAsync
        (DatabaseContext databaseContext, PaginationPage page = null,
        CancellationToken cancellationToken = default(CancellationToken))
    }
}

```



```

        {
            if (page == null) page = new PaginationPage();

            return await databaseContext.Cars
                .OrderBy(x => x.Guid).Skip((page.PageNumber - 1) * page.
                PageSize)
                .ToArrayAsync(cancellationToken);
        }

        /// <summary>
        /// the method returns a car by identifier
        /// </summary>
        /// <param name="identifier">car's identifier</param>
        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>car</returns>
        public async Task<ResponseWrapper<Car>> GetCarAsync
        (DatabaseContext databaseContext, Guid identifier, CancellationToken
        cancellationToken = default(CancellationToken))
        {
            var car = await databaseContext.Cars.Where(x => x.Guid ==
            identifier).FirstOrDefaultAsync(cancellationToken);

            return new ResponseWrapper<Car>
            {
                Status = car == null ? System.Net.HttpStatusCode.
                NotFound : System.Net.HttpStatusCode.OK,
                Content = car
            };
        }

        /// <summary>
        /// the method creates a new car
        /// </summary>
        /// <param name="car">car object</param>
        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>created car</returns>
        public async Task<ResponseWrapper<Car>> CreateCarAsync
        (DatabaseContext databaseContext, Car car, CancellationToken
        cancellationToken = default(CancellationToken))
        {
            if (!car.Guid.Equals(default(Guid))) return new
            ResponseWrapper<Car> { Status = System.Net.HttpStatusCode.Conflict };

            car.Guid = Guid.NewGuid();
            car.Created = DateTime.Now;

            databaseContext.Cars.Add(car);
            await databaseContext.SaveChangesAsync(cancellationToken);

            return new ResponseWrapper<Car> { Status = System.Net.

```

```

        HttpStatusCode.OK, Content = car };
    }

    /// <summary>
    /// the method updates an existing car
    /// </summary>
    /// <param name="car">car object</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>updated car</returns>
    public async Task<ResponseWrapper<Car>> UpdateCarAsync
(DatabaseContext databaseContext, Car car, CancellationTokens
cancellationToken = default(CancellationTokens))
    {
        var result = await GetCarAsync(databaseContext, car.Guid,
cancellationToken);
        if (!result.Succeeded) return result;

        var existing = result.Content;

        existing.End = car.End;
        existing.Power = car.Power;
        existing.ProducerGuid = car.ProducerGuid;
        existing.Series = car.Series;
        existing.Start = car.Start;
        existing.Type = car.Type;

        await databaseContext.SaveChangesAsync(cancellationToken);

        return new ResponseWrapper<Car> { Status = System.Net.
HttpStatusCode.OK, Content = existing };
    }

    /// <summary>
    /// the method deletes a car with the passed identifier
    /// </summary>
    /// <param name="guid">car's identifier</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>http status</returns>
    public async Task<ResponseWrapper> DeleteCarAsync
(DatabaseContext databaseContext, Guid identifier, CancellationTokens
cancellationToken = default(CancellationTokens))
    {
        var result = await GetCarAsync(databaseContext, identifier,
cancellationToken);
        if (!result.Succeeded) return new ResponseWrapper { Status =
result.Status };

        databaseContext.Cars.Remove(result.Content);
        await databaseContext.SaveChangesAsync(cancellationToken);
    }

```

```

        return new ResponseWrapper { Status = System.Net.
        HttpStatusCode.OK };
    }
}

```

CarService

Der CarService wiederum empfängt den Datenbank Kontext mittels DI und wird selbst den Controllern mittels DI bereitgestellt. Dazu muss dieser durch ein Interface gemäß SOLID principles definiert sein (vgl. .NET Core DI).

```

/// <summary>
/// the interface provides a generic template of a car service
/// </summary>
public interface ICarService
{
    /// <summary>
    /// the method returns all cars in accordance to the passed
    pagination page
    /// </summary>
    /// <param name="paginationPage">pagination page</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>set of cars</returns>
    public Task<ICollection<Car>> GetCarsAsync(PaginationPage
    paginationPage, Cancellation token cancellationToken = default
    (CancellationToken));

    /// <summary>
    /// the method returns a car by identifier
    /// </summary>
    /// <param name="identifier">car's identifier</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>car</returns>
    public Task<ResponseWrapper<Car>> GetCarAsync(Guid identifier,
    Cancellation token cancellationToken = default(CancellationToken));

    /// <summary>
    /// the method creates a new car
    /// </summary>
    /// <param name="car">car object</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>created car</returns>
    public Task<ResponseWrapper<Car>> CreateCarAsync(Car car,
    Cancellation token cancellationToken = default(CancellationToken));

    /// the method updates an existing car
    /// </summary>
    /// <param name="car">car object</param>

```

```

        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>updated car</returns>
        public Task<ResponseWrapper<Car>> UpdateCarAsync(Car car,
CancellationTokn cancellationToken = default(CancellationToken));

        /// <summary>
        /// the method deletes a car with the passed identifier
        /// </summary>
        /// <param name="guid">car's identifier</param>
        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>http status</returns>
        public Task<ResponseWrapper> DeleteCarAsync(Guid guid,
CancellationTokn cancellationToken = default(CancellationToken));
    }

    /// <summary>
    /// the service provides methods for car manipulation
    /// </summary>
    public class CarService : ICarService
    {
        /// <summary>
        /// ef core database context
        /// </summary>
        private DbContext _DbContext;

        /// <summary>
        /// the constructor creates a new instance of the car service
        /// </summary>
        /// <param name="dbContext">ef core database context<
/param>
        public CarService(DbContext dbContext)
        {
            _DbContext = dbContext;
        }

        /// <summary>
        /// the method returns all cars in accordance to the passed
pagination page
        /// </summary>
        /// <param name="paginationPage">pagination page</param>
        /// <param name="cancellationToken">cancellation token</param>
        /// <returns>set of cars</returns>
        public async Task<ICollection<Car>> GetCarsAsync(PaginationPage
paginationPage, CancellationToken cancellationToken = default
(CancellationToken))
        {
            return await new CarRepository().GetCarsAsync
(_DbContext, paginationPage, cancellationToken);
        }
    }

```

```

    /// <summary>
    /// the method returns a car by identifier
    /// </summary>
    /// <param name="identifier">car's identifier</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>car</returns>
    public async Task<ResponseWrapper<Car>> GetCarAsync(Guid
identifier, CancellationToken cancellationToken = default
(CancellationToken))
    {
        return await new CarRepository().GetCarAsync
(_DatabaseContext, identifier, cancellationToken);
    }

    /// <summary>
    /// the method creates a new car
    /// </summary>
    /// <param name="car">car object</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>created car</returns>
    public async Task<ResponseWrapper<Car>> CreateCarAsync(Car car,
CancellationToken cancellationToken = default(CancellationToken))
    {
        return await new CarRepository().CreateCarAsync
(_DatabaseContext, car, cancellationToken);
    }

    /// <summary>
    /// the method updates an existing car
    /// </summary>
    /// <param name="car">car object</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>updated car</returns>
    public async Task<ResponseWrapper<Car>> UpdateCarAsync(Car car,
CancellationToken cancellationToken = default(CancellationToken))
    {
        return await new CarRepository().UpdateCarAsync
(_DatabaseContext, car, cancellationToken);
    }

    /// <summary>
    /// the method deletes a car with the passed identifier
    /// </summary>
    /// <param name="guid">car's identifier</param>
    /// <param name="cancellationToken">cancellation token</param>
    /// <returns>http status</returns>
    public async Task<ResponseWrapper> DeleteCarAsync(Guid guid,
CancellationToken cancellationToken = default(CancellationToken))
    {
        return await new CarRepository().DeleteCarAsync

```

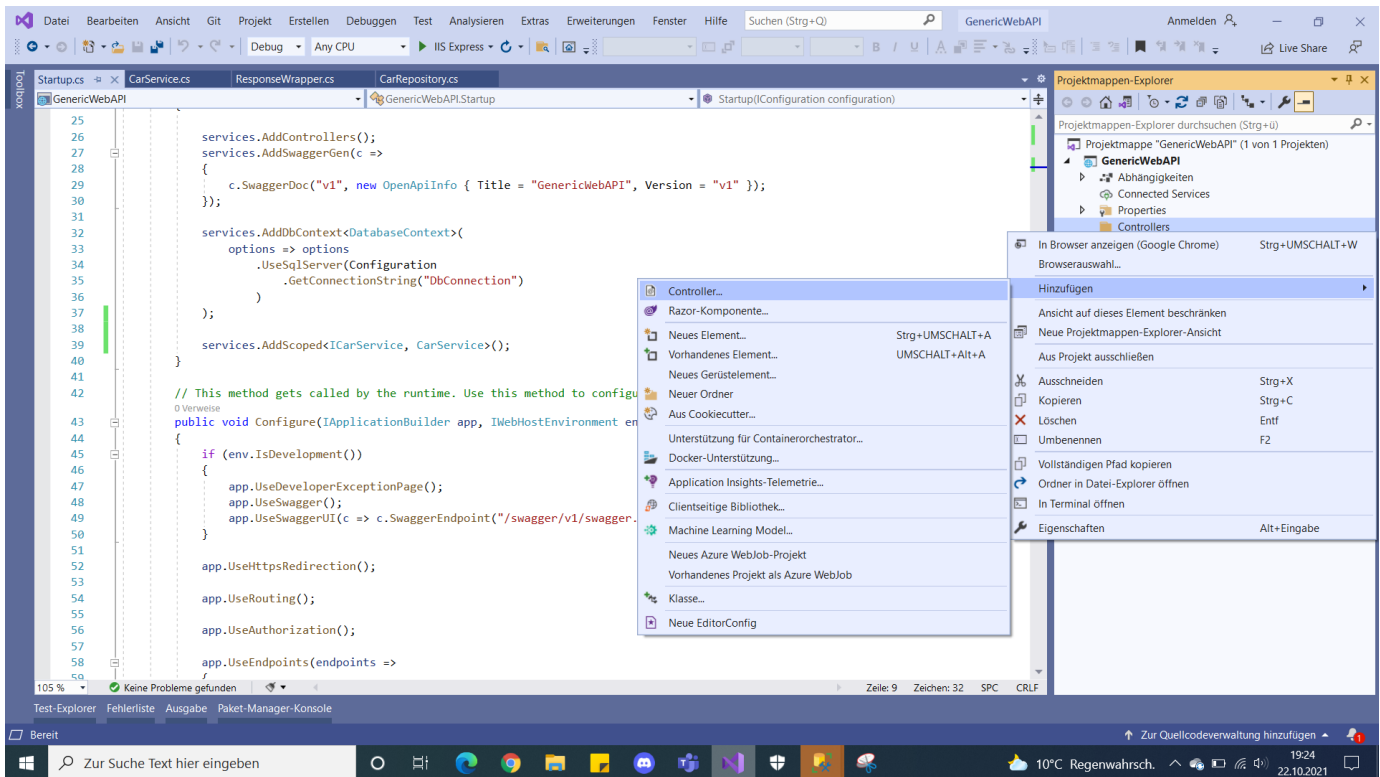
```
(_DatabaseContext, guid, cancellationTokens);
    }
}
```

Controller hinzufügen

Controller definieren die HTTP Schnittstelle. Als solche validieren diese die Autorisierung des Clients und geben den HTTP Status und ggf. Content zurück.

Um einen neuen Controller hinzuzufügen rechtsklicken Sie den Ordner Controller und wählen Hinzufügen Controller. Wählen Sie im Menü API API Controller leer. Ich nenne meinen Controller CarController.

NICHT MVC! Wir bauen eine RESTful API, keine **ASP.NET MVC** Webanwendung oder Razor Page.



Anschließend müssen im Controller die Endpunkte definiert werden, welche den Clients zur Verfügung gestellt werden sollen. Dabei müssen Endpunkte für alle CRUD Operationen definiert werden. Das HTTP Verb wird als Annotation zusammen mit der entsprechenden Route vorgesehen.

Bei allen Endpunkten, welche einen ResponseWrapper mit Content bereitstellen, muss, im Falle eines Errors, der Status Code und, im Falle von Erfolg, das Objekt zurückgegeben werden.

```
using GenericWebAPI.Models;
using GenericWebAPI.Models.Request;
using GenericWebAPI.Services;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Threading;
using System.Threading.Tasks;
```

```

namespace GenericWebAPI.Controllers
{
    [Route("api/car")]
    [ApiController]
    public class CarController : ControllerBase
    {

        private ICarService _CarService;

        public CarController(ICarService carService)
        {
            _CarService = carService;
        }

        [HttpPost("search")]
        public async Task<IActionResult> GetCarsAsync([FromBody]
PaginationPage page, CancellationToken cancellationToken)
        {
            return new OkObjectResult(await _CarService.GetCarsAsync
(page, cancellationToken));
        }

        [HttpGet("{identifier}")]
        public async Task<IActionResult> GetCarAsync([FromRoute] Guid
identifier, CancellationToken cancellationToken)
        {
            var result = await _CarService.GetCarAsync(identifier,
cancellationToken);

            if (!result.Succeeded) return new StatusCodeResult((int)
result.Status);

            return new OkObjectResult(result.Content);
        }

        [HttpPost("")]
        public async Task<IActionResult> CreateCarAsync([FromBody] Car
car, CancellationToken cancellationToken)
        {
            var result = await _CarService.CreateCarAsync(car,
cancellationToken);

            if (!result.Succeeded) return new StatusCodeResult((int)
result.Status);

            return new OkObjectResult(result.Content);
        }

        [HttpPut("")]
        public async Task<IActionResult> UpdateCarAsync([FromBody] Car

```

```

car, CancellationToken cancellationToken)
{
    var result = await _CarService.UpdateCarAsync(car,
cancellationToken);

    if (!result.Succeeded) return new StatusCodeResult((int)
result.Status);

    return new OkObjectResult(result.Content);
}

[HttpDelete("{identifizier}")]
public async Task<IActionResult> DeleteCarAsync([FromRoute]
Guid identifizier, CancellationToken cancellationToken)
{
    var result = await _CarService.DeleteCarAsync(identifizier,
cancellationToken);

    return new StatusCodeResult((int)result.Status);
}
}
}

```

Erstes Testen

Um die API erstmalig zu testen, erstellen wir ein neues xUnit Testprojekt im Projektordner. Dies wird erreicht, indem man die Projektkarte rechtsklickt und dann Hinzufügen -> Neues Projekt wählt. Anschließend sucht man einfach nach xunit, einem Testframework für Unit- und Integrationstests mit .NET.

The screenshot shows the 'Neues Projekt hinzufügen' (Add New Project) dialog in Visual Studio. The search bar at the top contains the text 'xunit'. Below the search bar, there are filters for 'C#' (language), 'Alle Plattformen' (all platforms), and 'Alle Projekttypen' (all project types). The search results are displayed in a list. The first result is 'xUnit-Testprojekt', which is highlighted. Below it is 'NUnit-Testprojekt'. Further down, under 'Andere Ergebnisse basierend auf Ihrer Suche' (Other results based on your search), there are more results for 'xUnit-Testprojekt' and 'NUnit-Testprojekt' for different languages like Visual Basic and F#. At the bottom right, there is a 'Weiter' (Next) button.

Zunächst muss ein Projektverweis auf unsere WebAPI angelegt werden. Rechtsklicken Sie dazu das Testprojekt Hinzufügen -> Projektverweis und wählen Sie die WebAPI.

Anschließend brauchen wir 2 Dateien im Testprojekt:

- Startup.cs Klasse
- Testklasse CarControllerTests.cs

✓ Übernehmen Sie gerne die unten aufgeführte Startup Klasse.

```
using GenericWebAPI.Controllers;
using GenericWebAPI.Helpers;
using GenericWebAPI.Services;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System.Reflection;

namespace GenericWebAPITests
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to
        add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers()
                .AddApplicationPart(typeof(CarController).GetTypeInfo().
Assembly)
                .AddControllersAsServices(); ;

            services.AddDbContext<DatabaseContext>(options => options.
UseSqlServer("Data Source=.\SQLEXPRESS;Initial Catalog=CarServiceData;
Trusted_Connection=True;MultipleActiveResultSets=true;"));

            services.AddTransient<ICarService, CarService>();
        }

        // This method gets called by the runtime. Use this method to
        configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
```

```

IWebHostEnvironment env)
{
    app.UseRouting();


    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}
}

```

Wichtig

Verweisen Sie mit der Startup Klasse auf die Klasse im Testprojekt und nicht auf die Startup Klasse des Liveprojekts!

 Installieren Sie ggf. notwendige Abhängigkeiten über NuGet. Am besten funktioniert dies, wenn Sie Errors hovern und den Quick Tipp im VS übernehmen. In der Regel werden fehlende Dependencies erkannt und deren Installation als Quick Tipp vorgeschlagen.

```

using GenericWebAPI.Models;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.TestHost;
using System.Net.Http;
using System.Text.Json;
using System.Threading.Tasks;
using Xunit;

namespace GenericWebAPITests
{
    public class CarControllerTests
    {
        private readonly TestServer _server;
        private readonly HttpClient _client;

        public CarControllerTests()
        {
            _server = new TestServer(new WebHostBuilder().
UseStartup<Startup>());
            _client = _server.CreateClient();
        }

        [Fact]
        public async Task TestCreateCar()
        {
            var serializedCar = JsonSerializer.Serialize(new Car {
                Type = CarType.Limousine,
                Power = 190,
            });
        }
    }
}

```

```

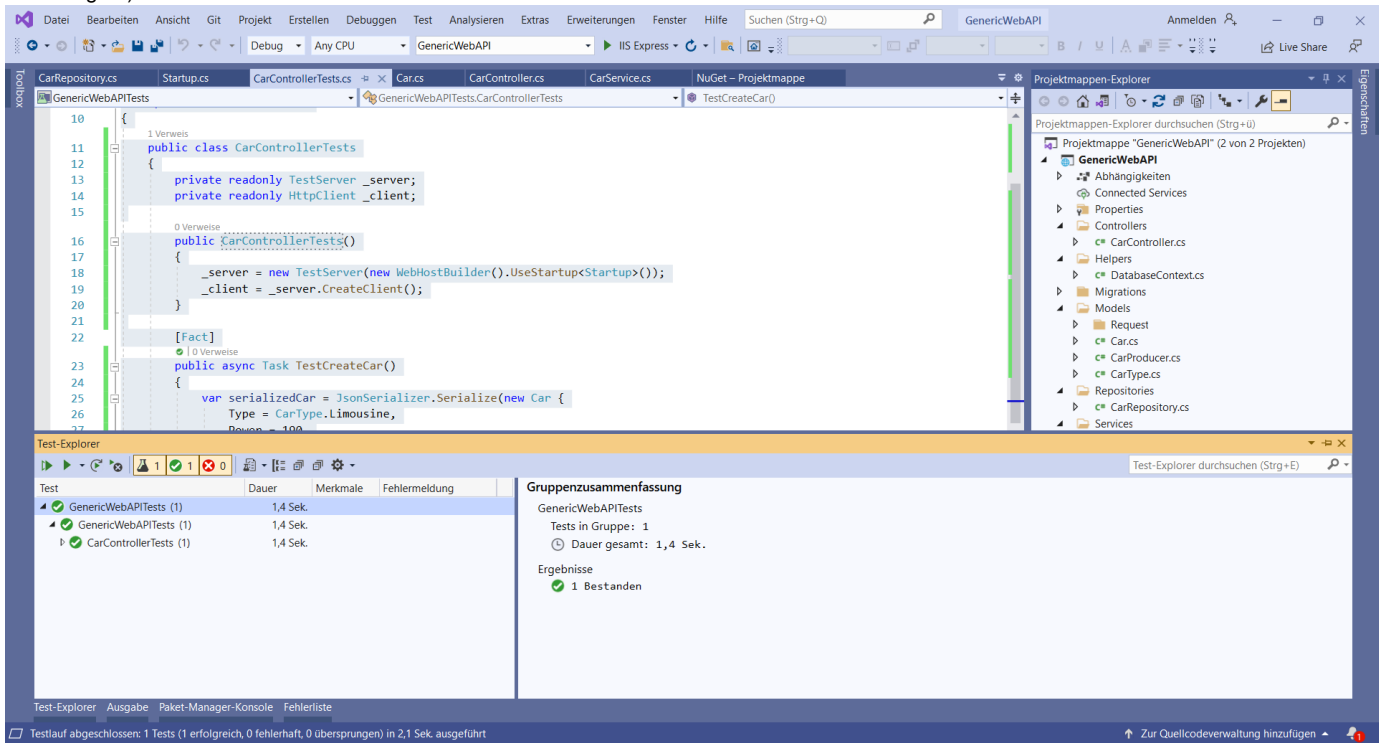
        Series = "3er Serie (330d)",
        Start = new System.DateTime(2012, 7, 1),
        End = new System.DateTime(2018, 10, 1),
    });

    var result = await _client.PostAsync("api/car", new
StringContent(serializedCar, System.Text.Encoding.Default, "application
/json"));

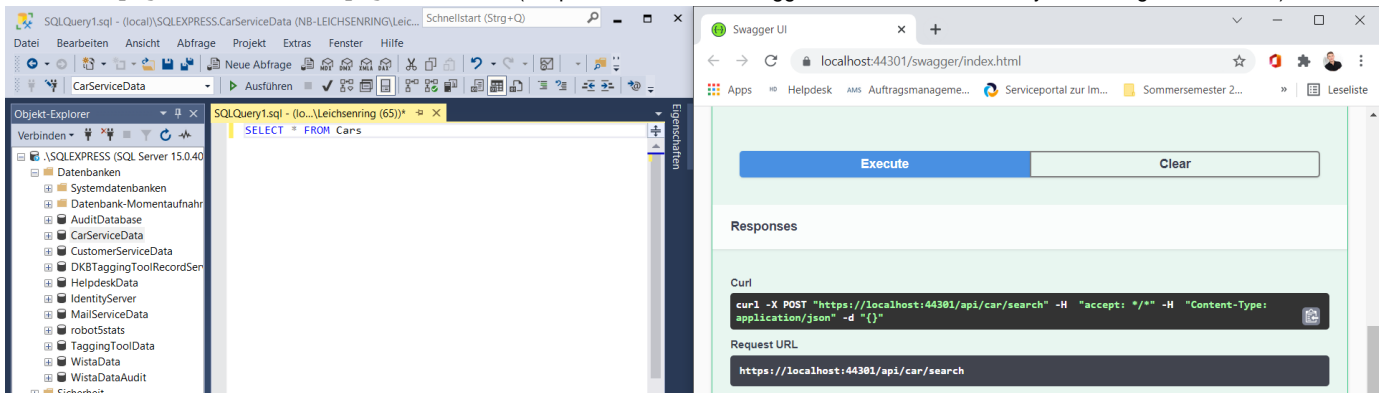
    Assert.True(result.IsSuccessStatusCode);
}
}
}

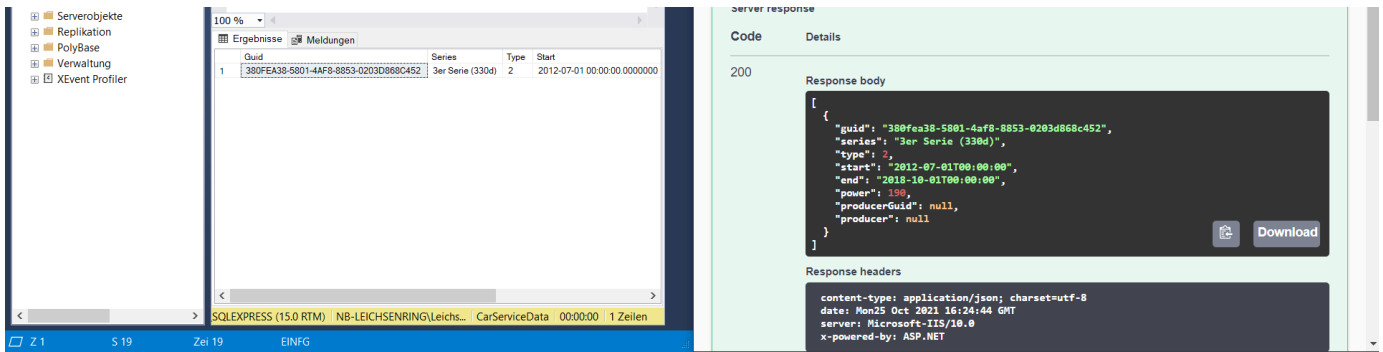
```

Anschließend können Sie den Test durchlaufen lassen. Wenn alles stimmt erscheint der Test im Test-Explorer als erfolgreich durchgeführt (vgl. Abbildung 12).



Als Resultat liegt jetzt ein erstes Fahrzeug (ohne Produzent) in der Datenbank (vgl. Abbildung 13, links). Diese können wir uns auch nach Starten des Hauptprojektes (Klick auf den Run-Button in der Navigationsleiste im Visual Studio) über die Swagger API Dokumentation anzeigen lassen (vgl. Abbildung 13, rec). Ich nutze dazu den Endpunkt `POST/search` des `CarControllers` und sende ein JSON Objekt als Request Body mit den Werten `pageNumber = 1` und `pageSize = 10` (Requests können bei Swagger über einen Klick auf Try it out ausgeführt werden).





Authentisierung und Autorisierung

Möchten Sie eine öffentliche API bereitstellen? Dann einfach skippen!

Interesse an einem eigenen Authentisierungsservice? Dann unbedingt hier einen Service raussuchen: <https://openid.net/developers/certified/>

Im Fall von .NET empfehle ich den IDS4 Service. Abgesehen davon, dass dieser alles enthält, was man braucht und sich zusätzlich ganz hervorragend an .NET Services anbinden lässt, ist dieser sehr gut dafür geeignet, um die Client Credentials und Code Grant Flows zu erlernen und zu verstehen.

Nachdem wir Daten in unserer Datenbank haben und diese auch mit einer RESTful API abholen können, müssen wir nun den Zugang zu diesen absichern. Dazu nutzen wir den OpenID Provider Identity Server 4, welchen wir selbst zu unseren Zwecken hosten.

Zunächst müssen wir definieren, welche Policies (Richtlinien) wir in unserer App vorsehen. Dazu nutze ich einen Policy Builder:

```

using Microsoft.AspNetCore.Authorization;
using System;
using System.Collections.Generic;

namespace GenericWebAPI.Helpers
{
    /// <summary>
    /// the class provides all application policies
    /// </summary>
    public static class PolicyStore
    {
        /// <summary>
        /// policy collection
        /// </summary>
        public static class Policies
        {
            public static int AdministrativeAccount = 0;
        }

        /// <summary>
        /// policy builders
        /// </summary>
        public static Dictionary<int,
Action<AuthorizationPolicyBuilder>> PolicyBuilders = new
Dictionary<int, Action<AuthorizationPolicyBuilder>>
  
```

```

        {
            { Policies.AdministrativeAccount, policy =>
                {
                    policy.RequireAuthenticatedUser();
                    policy.RequireClaim(System.Security.Claims.
ClaimTypes.Role, "Administrator");
                }
            }
        };
    }
}

```

Wie wir sehen kennt unsere App nur eine Richtlinie namens administrative Accounts.

Anschließend definieren wir diese Richtlinie und unseren Identity Server in der Startup Klasse (vgl. Abbildung 14). Wichtig ist dabei, dass wir unsere Authority in den appsettings.json entsprechend der Benennung in der Methode AddIdentityServerAuthentication hinterlegen.

```

// This method gets called by the runtime. Use this method to add services to the container.
0 Verweise
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "GenericWebAPI", Version = "v1" });
    });

    services.AddDbContext<DatabaseContext>(
        options => options
            .UseSqlServer(Configuration.GetConnectionString("DbConnection"))
    );

    services.AddTransient<ICarService, CarService>();

    services
        .AddAuthentication(IdentityServerAuthenticationDefaults.AuthenticationScheme)
        .AddIdentityServerAuthentication(options =>
        {
            options.Authority = Configuration.GetValue<string>("OidcAuthority");
            options.EnableCaching = true;
        });

    services.AddAuthorization(options =>
    {
        options.AddPolicy(nameof(PolicyStore.Policies.AdministrativeAccount), x => PolicyStore.PolicyBuilders[PolicyStore.Policies.AdministrativeAccount](x));
    });
}

```

Schließlich müssen wir noch in der Methode Configure die Methode app.UseAuthorization() hinzufügen. Anschließend sieht unsere Startup.cs wie folgt aus:

```

using GenericWebAPI.Helpers;
using GenericWebAPI.Services;
using IdentityServer4.AccessTokenValidation;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.OpenApi.Models;

namespace GenericWebAPI
{
    public class Startup

```

```

{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to
    add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title =
"GenericWebAPI", Version = "v1" });
        });

        services.AddDbContext<DatabaseContext>(
            options => options
                .UseSqlServer(Configuration.GetConnectionString
("DbConnection"))
        );

        services.AddTransient<ICarService, CarService>();

        services
            .AddAuthentication(IdentityServerAuthenticationDefaults.
AuthenticationScheme)
            .AddIdentityServerAuthentication(options =>
            {
                options.Authority = Configuration.GetValue<string>
("OidcAuthority");
                options.EnableCaching = true;
            });

        services.AddAuthorization(options =>
        {
            options.AddPolicy(nameof(PolicyStore.Policies.
AdministrativeAccount), x => PolicyStore.PolicyBuilders[PolicyStore.
Policies.AdministrativeAccount](x));
        });
    }

    // This method gets called by the runtime. Use this method to
    configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)

```

```

{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1
/swagger.json", "GenericWebAPI v1"));
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}
}

```

Jetzt ist unser Service bereit, JWT Tokens zu verarbeiten und gegen den Introspection Endpoint des IDS zu prüfen. Um eine Methode abzusichern, verwenden wir die `[Authorize(Policy = 'XYZ')]` Annotation über dem jeweiligen Endpunkt. In Abbildung 14 wird demonstriert, wie der Endpunkt `search` mit der Policy `AdministrativeAccount` abgesichert wird und eine beispielhafte Abfrage über Swagger mit einer 401 Unauthorized beantwortet wird. Dies passiert, da wir keinen Authorization Header senden.

The screenshot shows two windows. On the left is the Visual Studio code editor with the `CarController.cs` file open. The `search` endpoint is highlighted, which is annotated with `[Authorize(Policy = nameof(PolicyStore.Policies.AdministrativeAccount))]`. On the right is the Swagger UI browser window at `localhost:44301/swagger/index.html`. The `search` endpoint is selected, and the 'Execute' button has been clicked. The response shows a 401 Unauthorized error with the message 'Error: 401 Unauthorized'.

Wenn wir uns jetzt einen Token vom IDS abholen und diesen zusammen mit der Abfrage senden, dann erhalten wir wieder einen Erfolg als Resultat (vgl. Abbildung 15). Im Test wird Postman als HTTP Client verwendet.

NewImportRunner

My Workspace

Invite

Filter

HistoryCollectionsAPIs

+ New Collection

test
1 request

POST https://localhost:44301/api/car/search

SendSave

ParamsAuthorizationHeaders (9)BodyPre-request ScriptTestsSettings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

BodyCookiesHeaders (5)Test Results

Status: 200 OKTime: 195 msSize: 374 BSave Response

PrettyRawPreviewVisualizeJSON

```
1 {
2   "guid": "380fea38-5801-4af8-8853-0203d868c452",
3   "series": "3er Serie (330d)",
4   "type": 2,
5   "start": "2012-07-01T00:00:00",
6   "end": "2018-10-01T00:00:00",
7   "power": 190,
8   "producerGuid": null,
9   "producer": null
10 }
11
12
```

Find and ReplaceConsole

Bootcamp