Einführung in WebAPI

Was ist eine API?

API steht für Application Programming Interface. Es ist eine Schnittstelle, die es ermöglicht, dass zwei Programme miteinander kommunizieren können.

WEB API

Im web werden APIs verwendet, um Daten zwischen dem Server und dem Client auszutauschen. Wir kennen aus letzem jahr als wir einfach nur den "Normalen" Austausch von Daten zwischen Server und Client also das der Server alle benötigten Daten an den Client schickt und der Client diese dann anzeigt. Das heißt, dass der Server die ganze HTML, CSS und JavaScript Dateien an den Client schickt. Das ist aber nicht immer die beste Lösung, da es sehr viel Datenverkehr gibt und die Seite sehr langsam wird.

Deshalb gibt es die Web API, die es ermöglicht, dass der Server nur die Daten an den Client schickt, die der Client benötigt. Der Client kann dann die Daten anzeigen. Dies geschiet meistens in JSON oder XML Format.

WEB API in ASP.NET

ASP.NET Web API ist ein Framework, das es ermöglicht, HTTP-Dienste zu erstellen, die von einer Vielzahl von Clients wie Browsern und mobilen Geräten verwendet werden können. Aktuell Programmieren wir nur das Backend, also wir designen kein Frontend (HTML, CSS, JavaScript). Wir schreiben nur die API, die die Daten an den Client schickt.

HTTP Methoden

HTTP Methoden sind die Methoden, die es ermöglichen, dass der Client mit dem Server kommunizieren kann. Es gibt 5 Methoden, die am häufigsten verwendet werden:

- GET -> wir bekommen die Daten
- POST -> wir schicken die Daten
- PUT -> wir ändern die Daten

- DELETE -> wir löschen die Daten
- PATCH -> wir ändern die Daten

Midelware

Middleware ist eine Software, die zwischen zwei oder mehreren Programmen oder Anwendungen vermittelt. In ASP.NET Core Middleware ist eine Software, die zwischen dem Client und dem Server vermittelt. Sie kann die Anforderung des Clients bearbeiten und die Antwort des Servers bearbeiten. Sie ist also essenziell für die Web API und die Kommunikation zwischen Client und Server.

Swager

Ist ein praktisches Tool extra für die EntwicklerInnen, um die API zu testen ohne eine extra Software zu installieren bzw keine extra anwendung zu schreiben.

Swager erzeugt eine Weboberfläche, auf der man alle funktionen der API testen kann.

Wie baut man eine URL für eine API auf?

Beispiel:

```
https://api.example.com/api/shops/articles + GET
                                /articles/3 + GET
                                /articles/2 + DEL
                                /articles + PUT
                                /articles + POST
                                /articles?api_key=1234 + GET
```

Der Anfang der URL ist immer die Domain das wir wissen auf welcher Seite wir uns befinden und welchen Server wir ansprechen. Dann kommt der Pfad, der angibt, welche Daten wir haben wollen. Dann kommt die Methode, die angibt, was wir mit den Daten machen wollen.
Es gibt nicht wirkliche Richtlinien, wie man die URL aufbauen soll, es ist allerdings ratsam, dass man den Pfad logisch bennent und mit /api/ anfäng und dann schreibt welche Daten man benötigt.

Was benötigen wir um Web API zu Programmieren?

- Alle benötigten NuGet-Pakete also ASP.NET Core Web API, Entity Framework Core und eine verbingung zur Datenbank.
- Wir müssen als erstes Daten haben auf die wir zugreifen wollen -> eine Klasse aus der wir eine Datenbanktabelle erstellen können.
- Wir müssen eine Datenbank erstellen (es empfiehlt sich Testdaten zu haben)
- Wir benötigen eine Klasse, die die Datenbankverbindung herstellt (DbManager)
- Wenn alles eingerichtet ist, können wir die API schreiben.
- Dafür benötigen wir eine Klasse, die die API Kontrolliert (Controller)
- In dieser Klasse schreiben wir die Methoden, die die Daten aus der Datenbank holen und an den Client schicken.
- Wenn alle Methoden geschrieben sind, können wir die API testen.

API Controller

Ein Controller ist eine Klasse, die die API kontrolliert. In dieser Klasse schreiben wir die Methoden, die die Daten aus der Datenbank holen und an den Client schicken.

Beispiel:

```
//Alles was vor der Klasse definiert ist//

//Hier wird definiert wie wir auf die Daten zugreifen also die URL unter der wir die ganze Klass
[Route("api/[controller]")]
[ApiController]
public class ShopController : ControllerBase
{
    //Hier wird die Datenbankverbindung hergestellt
    private readonly DbManager _dbManager;

    public ShopController(DbManager dbManager)
    {
        _dbManager = dbManager;
    }

    //Hier wird definiert wie wir auf die Daten zugreifen also die URL unter der wir die Methode
    //Hier wird definiert, dass wir die Daten nur mit GET bekommen
    //GET -> wir bekommen die Daten
    [HttpGet("articles")]
    public async Task<ActionResult<IEnumerable<Article>>> GetArticles()
    {
        return await _dbManager.GetArticles();
    }

    //Hier können wir einen Artikel mit einer bestimmten ID bekommen
    //durch die Geschwungene Klammern wird die ID dynamisch -> wir können jede ID eingeben
    [HttpGet("articles/{id}")]
    public async Task<ActionResult<Article>> GetArticle(int id)
    {
        return await _dbManager.GetArticle(id);
    }

    //Hier können wir einen Artikel löschen
    //durch die Geschwungene Klammern wird die ID dynamisch -> wir können jede ID eingeben
    [HttpDelete("articles/{id}")]
    public async Task<ActionResult<Article>> DeleteArticle(int id)
    {
        var article = await _dbManager.GetArticle(id);  //Hier wird der Artikel mit der ID geho
        if (article == null)
        {
            return NotFound();
        }
        var ergebnis = await _dbManager.DeleteArticle(article); //Hier wird der Artikel gelöscht
```

```
        await _dbManager.SaveChangesAsync();    //Hier wird die Änderung in der Datenbank gespei
        return newJsonResult(ergebnis); //Hier wird das Ergebnis zurückgegeben -> true oder fals
    }
    //einen neuen Artikel erzeugen
        [HttpPost("article")]
        public async Task<IActionResult> AddArticle(Article article, string apiKey){
            var erfolg = await _dbManager.Articles.AddAsync(article);   //Hier wird der Artikel
            await _dbManager.SaveChangesAsync();                //Hier wird die Änderung in der
            return new JsonResult(erfolg);                    //Hier wird das Ergebnis zurückgeg
        }

        //einen anderen Artikel ändern
        [HttpPut("article")]
        public async Task<IActionResult> UpdateArticle(Article article, string apiKey){
            var articleToUpdate = await _dbManager.Articles.FindAsync(article.ArticleId);   //Hi
            articleToUpdate.Name = article.Name;                //Hier wird der Name geändert
            articleToUpdate.Price = article.Price;          //Hier wird der Preis geändert
            articleToUpdate.ReleaseDate = article.ReleaseDate;      //Hier wird das Release Date
            var erfolg = _dbManager.Articles.Update(articleToUpdate);       //Hier wird der Arti
            await _dbManager.SaveChangesAsync();            //Hier wird die Änderung in der Date
            return new JsonResult(erfolg);          //Hier wird das Ergebnis zurückgegeben -> tr
        }

}
```

API Key

Ein API Key ist ein Schlüssel, der es ermöglicht, dass der Client auf die API zugreifen kann und die Daten bekommt. Der API Key wird meistens in der URL mitgegeben und wird dann in der API überprüft. Wenn der API Key nicht stimmt, bekommt der Client keine Daten. Der API Key ist also eine Art Passwort, das der Client benötigt, um auf die API zugreifen zu können.

Der API key wird mittels eines Query Parameters übergeben. Ein Query Parameter ist ein Parameter, der an die URL angehängt wird. Der Query Parameter wird mit einem Fragezeichen (?) an die URL angehängt und besteht aus einem Schlüssel-Wert-Paar. Der Schlüssel ist der Name des Parameters und der Wert ist der Wert des Parameters. Der Schlüssel und der Wert werden mit einem Gleichheitszeichen (=) getrennt. Mehrere Parameter werden mit einem Kaufmanns-Und-Zeichen (&) getrennt.

In ASP.NET Core wird der API Key meistens in der URL mitgegeben. Das sieht dann so aus:

```
[HttpPost("article")]
public async Task<IActionResult> AddArticle(Article article, string apiKey)
{
    if (apiKey != "1234")
    {
        return Unauthorized();
    }
    var erfolg = await _dbManager.Articles.AddAsync(article);
    await _dbManager.SaveChangesAsync();
    return new JsonResult(erfolg);
}
```

In diesem Beispiel wird überprüft, ob der API Key 1234 ist. Wenn der API Key nicht 1234 ist, bekommt der Client keine Daten.
In der Realität wird eine Datenbanktabelle erstellt, in der die API Keys gespeichert sind. Wenn der API Key in der Datenbanktabelle ist, bekommt der Client die Daten. Wenn der API Key nicht in der Datenbanktabelle ist, bekommt der Client keine Daten.

Client Seite (Abfrage der API)

Die Client-Seite ist die Seite, die die Daten von der API bekommt und anzeigt. Die Client-Seite kann in verschiedenen Programmiersprachen geschrieben werden, z.B. in HTML, JavaScript aber auch in C#.

Wir benötigen um Daten ber HTTP abzufragen eine HTTPClient Klasse. Diese Klasse ermöglicht es uns, dass wir die Daten von der API abfragen können.

In diesem Beispiel wird die Client-Seite in C# geschrieben. In der Client-Seite wird die API abgefragt und die Daten angezeigt.

Einrichtung des HTTPClient:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

namespace Client
{
    class Program
    {
        //Hier wird der HTTPClient initialisiert
        private static HttpClient _articleClient = new HttpClient();

        static async Task Main(string[] args)
        {
            using (var client = new HttpClient())
            {
                //die Base Adresse der API wird bei der Initialisierung des HTTPClient angegeber
                client.BaseAddress = new Uri("https://localhost:5001/");

                HttpResponseMessage response = await client.GetAsync("api/shop/articles");
                //Mit einem API Key würde die URL so aussehen: api/shop/articles?apiKey=1234
                if (response.IsSuccessStatusCode)   //Hier wird überprüft ob die Anfrage erfolgr
                {
                    var articles = await response.Content.ReadAsAsync<IEnumerable<Article>>();
                    foreach (var article in articles)  //Hier werden die Daten durchgegangen
                    {
                        Console.WriteLine(article.Name);  //Hier wird der Name des Artikels ausg
                    }
                }
            }
        }
    }

}
```

Dependency Injection

Dependency Injection (DI) ist ein Designprinzip, das die Abhängigkeiten von Klassen verwaltet und die Erstellung und Bindung dieser Abhängigkeiten vereinfacht. In **ASP.NET Core** ist DI ein zentrales Konzept, das die Entwicklung modularer und testbarer Anwendungen unterstützt.

Vorteile der Dependency Injection

- **Entkopplung von Klassen**: Klassen sind nicht mehr direkt voneinander abhängig, sondern erhalten ihre Abhängigkeiten von einem externen Dienst.
- **Testbarkeit**: Klassen können leichter getestet werden, da ihre Abhängigkeiten durch Mock-Objekte ersetzt werden können.
- **Wiederverendbarkeit**: Abhängigkeiten können in verschiedenen Klassen wiederverwendet werden.
- **Erweiterbarkeit**: Neue Abhängigkeiten können einfach hinzugefügt werden, ohne den Code zu ändern.

Implementierung in ASP.NET Core

In **ASP.NET Core** wird Dependency Injection durch den `IServiceCollection` und den `IServiceProvider` realisiert. Die `IServiceCollection` ist eine Sammlung von Diensten, die in der Anwendung verwendet werden. Die `IServiceProvider` ist ein Dienstanbieter, der die Dienste bereitstellt.

Beispiel:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Registrierung von DbManager als Singleton
        services.AddSingleton<DbManager>();

        // Registrierung von ShopController als Transient
        services.AddTransient<ShopController>();

        // Weitere Service-Registrierungen
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

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

Verwendung im Controller

```
use Microsoft.Extensions.DependencyInjection;
//...
[Route("api/[controller]")]
[ApiController]
public class ShopController : ControllerBase
{
    private readonly DbManager _dbManager;

    public ShopController(DbManager dbManager)
    {
        _dbManager = dbManager;
    }

    // Methoden des Controllers
}
```

Durch die Verwendung von Dependency Injection wird die Erstellung und Verwaltung von Abhängigkeiten in ASP.NET Core vereinfacht und die Testbarkeit und Wartbarkeit der Anwendung verbessert.