APIer i .NET

HttpClient

- HttpClient är en klass i .NET som vi kan använda för att göra anrop till externa tjänster med hjälp av HTTP.
- Den hjälper oss alltså att bygga Requests och invänta och tolka ett Response.
- Detta är alltså vår motsvarighet till Fetch eller Axios.
- Men den möjliggör väldigt mycket mer.

HttpClient - riktlinjer

- Vi bör inte skapa flera instanser av HttpClient
  - Detta beror på att en HttpClient håller i en stor mängd information och anslutningsinställningar som bör återanvändas.
  - Om vi skapar för många instanser av HttpClient på kort tid kan det skapa problem pga begränsningar i http-protokollet.
- Om vi behöver flera olika HttpClient, tex till olika hosts så bör vi använda oss utav IHttpClientFactory.
  - IHttpClientFactory låter oss konfigurera flera möjliga klienter som vi sedan kan skapa för användning i begränsat scope.
  - Mer om denna nästa vecka.

HttpClient Exempel

```
var client = new HttpClient();
client.BaseAddress = new Uri("https://opentdb.com/");
var uri = "api.php?amount=10&category=18&difficulty=easy&type=boolean";
var response = await client.GetAsync(uri);
var body = await response.Content.ReadAsStringAsync();
```

- BaseAddress anger Host-adressen för HttpClient
- Metoden GetAsync kan användas för att skicka en GET-Request till adressen specificerad.
- Content innehåller bodyn av responsen som en dataström. Här används ReadAsStringAsync för att konvertera innehållet till string.

Dotnet-CLI

- Visual Studio hjälper oss med väldigt många saker, från att skapa projekt med mallar, installera nuget-paket, kraftfull debugger och visualisera olika saker.
- Ibland finns det behov av att bedriva .NET-utveckling utan möjlighet att använda Visual Studio.
- I dessa lägen kan vi fortfarande nyttja styrkan i mallar och nuget genom dotnet-cli

DEMO

Övning – 30 minuter

- Öppna en tom mapp i VisualStudio Code
- Öppna terminalen och skriv in `dotnet --version`
- Du bör då få upp ett versionsnummer typ 8.0.100
- Skriv nu `dotnet new console`
- Detta kommer att sätta upp ett konsollprojekt i den mappen du befinner dig.
- I Program.cs, instansiera en HttpClient, sätt BaseAddress till den host du använde i förra övningen.
- Anropa GetAsync till en av de paths du identifierade i förra övningen
- Hämta sedan innehållet i Body och skriv ut det i konsollen.

Vad är ASP.NET Core?

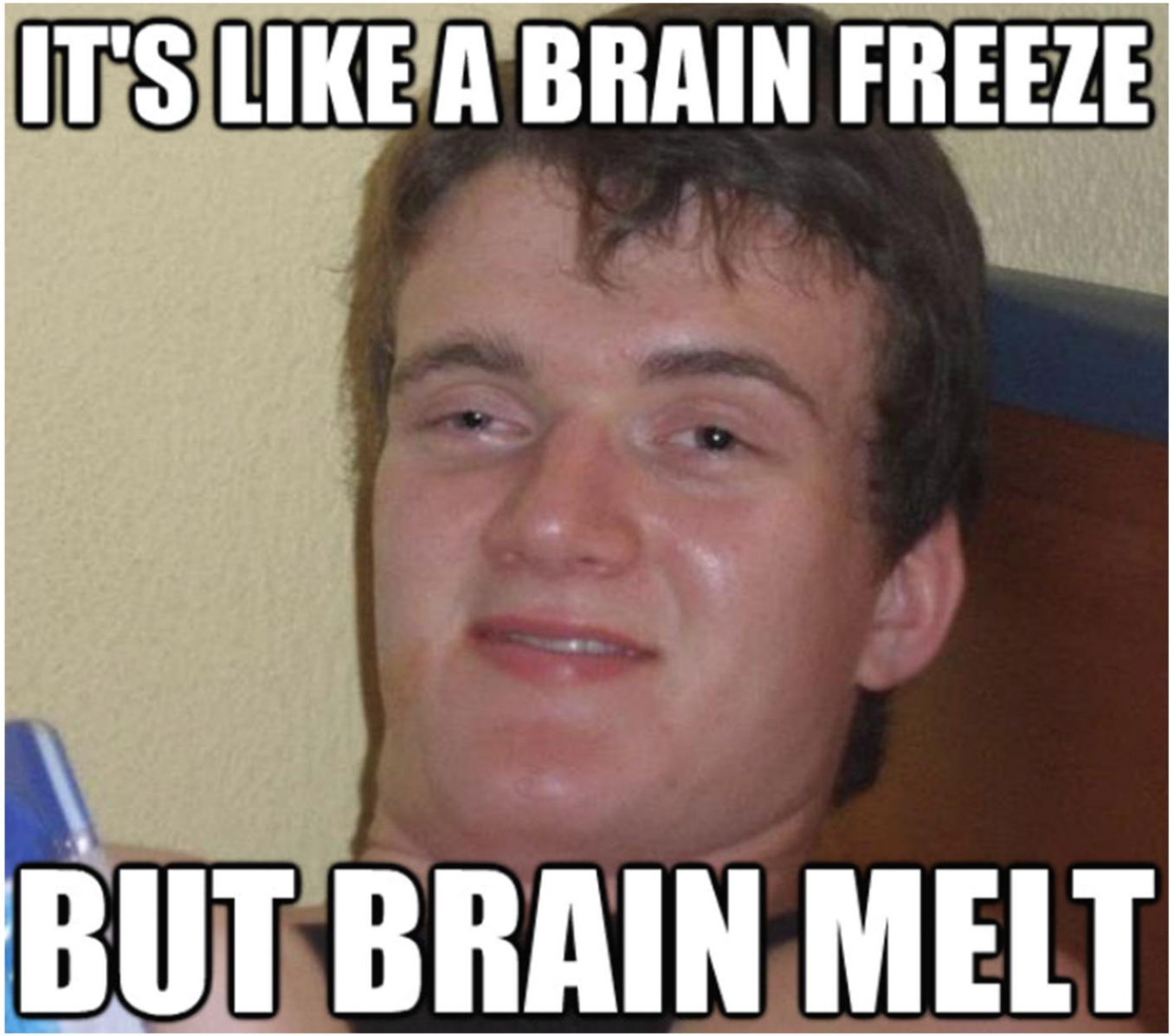
- ASP.NET, eller ASP.NET Core (samma sak numera) är en del av .NET som innehåller allt som behövs för att skapa webb-applikationer i .NET.
- ASP.NET används ofta synonymt med .NET och är den absolut mest använda delen av .NET tillsammans med LINQ och EntityFramework.
- Så fort vi vill lyssna efter något över internet så använder vi ASP.NET
- ASP.NET innehåller också många andra paket som är användbara när vi ska skapa API:er och andra webbtjänster

ASP.NET Basics

- En ASP.NET-applikation nyttjar framförallt tre s.k. Designmönster.
  - Builder-Pattern
  - Dependency Injection
  - Strategy-pattern
- Vi behöver inte förstå dessa mönster på djupet för att arbeta med ASP.NET men en grundläggande förståelse hjälper mycket.
- I kursen ”Clean Code och Testbar Kod” kommer ni att få en djupare förståelse för designmönster.

Hur ser en ASP.NET applikation ut?

- I en ASP.NET-applikation finns alltid någon form av applikationsobjekt som vi först bygger upp med det vi vill ha (Services, konfigurationer etc.) följt av något som kallas en ”Middleware pipeline”.
- Tidigare gjordes detta i en klass som hette Startup.cs. Numera görs allt i Program.cs
- I en ”Middleware pipeline” kan vi koppla på ytterligare tjänster och funktionalitet vi vill att vår applikation ska kunna nyttja/utföra. Tex. Definiera endpoints.


IT'S LIKE A BRAIN FREEZE
BUT BRAIN MELT


BUT WAIT
THERE'S MORE
memegenerator.n

WebApplication

- Som tidigare nämnt så bygger en ASP.NET-applikation på ett applikationsobjekt. Det första och enklaste i ASP.NET är WebApplication.
- WebApplication möjliggör bland annat att applikationen lyssnar på http-requests.
- Det låter oss också definiera endpoints och sätta upp loggning, säkerhet, databashantering och mycket mer.

De tre grundstegen i en ASP.NET-app

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

var app = builder.Build();

app.Run();
```

- Variabeln builder innehåller det applikationsobjekt som ska byggas upp.
- App innehåller det byggda objektet
- App.Run startar applikationen

DEMO

Övning 2 – Skapa och anropa ett ASP.NET-api

- Öppna en tom mapp i VS Code
- Öppna terminalen och kör kommandot `dotnet new web`
- Kör sedan kommandot `dotnet run`
- Kopiera adressen som syns i konsollen
- Öppna Postman och skicka en GET-Request

```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5227
```

- `app.MapGet("/", () => "Hello World!");`

  Definierar alltså en endpoint för pathen ”/” och metoden GET

- Ovanför builder.Build(), deklarera och instansiera en List<string> som innehåller orden ”Hund”, ”Katt”, ”Kanin” och returnera denna istället för ”Hello World!” från er GET ”/”-endpoint.
- Anropa endpointen i Postman.

DEMO

What about POST,PUT,PATCH,DELETE?

- De andra vanliga http-metoderna går givetvis också att definiera.
- För detta används:
  - MapPost
  - MapPut
  - MapPatch
  - MapDelete
- MapDelete kräver/förväntar sig ingen body eller annan data
- POST,PUT,PATCH gör det eftersom det är Create och Update-metoder

QueryParams, Body

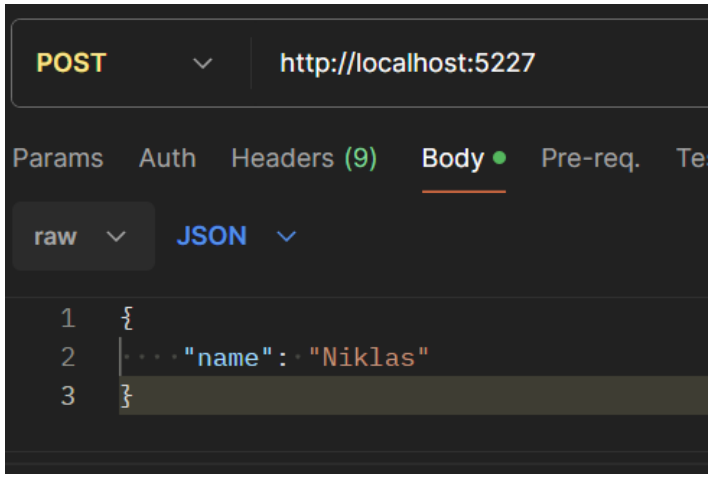
- För att ta emot Body och Query-parametrar behöver vi bara deklarera en parameter i funktionen som körs vid requesten med matchande namn och typ.

```
app.MapGet("/", (string hej)
```

```
?hej=WOOOOT
```

- Det samma gäller Body men då som en klass

```
app.MapPost("/", (Person person)
record Person(string Name);
```



Path-parametrar

- Ibland vill vi inte använda varken body eller query-parametrar men ändå skicka med någon form av id eller förändringsbar path.
- Då används s.k. path-variabler eller path-parametrar.
- Dessa deklareras i pathen med { } och i parameterlistan för funktionen med samma namn.

```
app.MapGet("/{id}", (int id)
```

```
http://localhost:5227/1
```

DEMO

Övning 3 – Post och Get

- Utgå från övningen innan.
- Lägg till en MapPost som tar emot en sträng och lägger till den i din lista.
- Lägg till en MapDelete som tar emot en integer och tar bort den strängen med id samma som det tal som skickats in.
- Lägg till en MapPut tar emot en integer och en string och som byter ut värdet på det indexet som skickas in med den strängen som skickats in.
- Testa samtliga endpoints i Postman

DEMO

Övning 4

- Testa att kombinera det vi gått igenom idag och återskapa övningen från i måndags fast i .NET.
- Alltså skapa en get-endpoint som anropar det öppna apiet och skriver ut svaret i konsollen.

Dependency Injection

- I en ASP.NET har vi möjligheten att lägga till servicar och annat som ska finnas tillgängligt under applikationens livstid (så länge den körs)
- Detta sker genom Dependency Injection
- En WebApplication builder innehåller något som kallas för ServiceColleciton. Denna agerar som en lista av allt det vi vill ha tillgång till.
- När vi lägger till servicar i ServiceCollection kan vi göra det med tre olika livslängder.
  - Transient
  - Scoped
  - Singleton

Vad betyder dessa livslängder?

- **Transient** innebär att varje gång servicen hämtas så får vi en ny instans av den.

```
builder.Services.AddTransient<PeopleStorage>();
```

- **Scoped** innebär att så länge en service används så får vi samma instans överallt vi begär den. (Detta är den vi använder mest)

```
builder.Services.AddScoped<PeopleStorage>();
```

- **Singleton** är som namnet antyder en livslängd som gör att vi endast har en och alltid en instans utav servicen.

```
builder.Services.AddSingleton<PeopleStorage>();
```

- En Service kan bara läggas till en gång. Alltså kan vi inte ha samma service med olika livslängder.

DEMO


THAT'S ALL,
FOLKS.