

# Praktische Informatik

Vorlesung 13

Netzwerkprogrammierung

# Zuletzt haben wir gelernt...

- Wie man Multithreading in C# umsetzt.
- Wie man Abhängigkeiten zwischen Threads organisiert.
- Warum und wie man Threads mit Monitoren und Semaphoren synchronisiert.
- Wie das Erzeuger-Verbraucher-Problem gelöst werden kann.
- Wie man Threads und WPF miteinander in Einklang bringt.
- Wie die Klasse Task aus der Task Parallel Library die Arbeit mit Gleichzeitigkeit vereinfacht.
- Wie man mit Hilfe von async und await asynchrone Programmierung umsetzt.

# Inhalt heute

- Sockets
- Client-Server-Kommunikation
- Die Klasse TcpClient
- Zeitclient und -Server
- E-Mails über SMTP versenden
- Web-Ressourcen und HttpClient
- Speisenchecker
- Restful-Services
- JSON
- OpenLigaDB

# Voraussetzungen

- Diese Vorlesung befasst sich mit der Umsetzung von Netzwerkkommunikation mit Hilfe von C# und dem .Net Framework.
  - Dabei werden verschiedene Kenntnisse vorausgesetzt.
- Was man schon verstanden haben sollte:
  - Netzwerke und der Datenaustausch darüber.
  - Das OSI-Schichtenmodell und der Zweck von Protokollen.
  - Die Begriffe Client, Server, Socket und Port.
  - Kommunikation im Internet: IP-Adressen, Routing.
  - Das TCP- und das UDP-Protokoll.
  - Protokolle der Anwendungsschicht: http, ftp, ...

# Netzwerkcommunication in C#

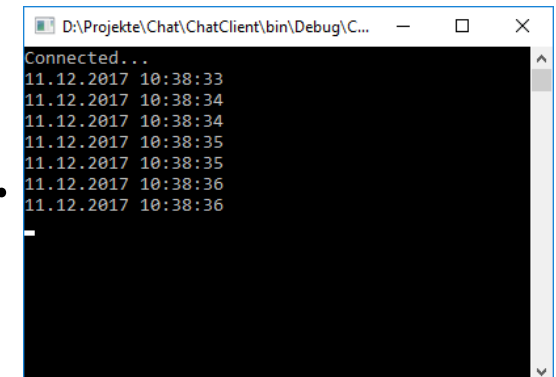
- Das .Net Framework stellt viele Klassen bereit, um Daten über Netzwerke zu versenden.
  - Die Klassen sind im Namespace System.Net zu finden.
- Dabei kann die Kommunikation auf einem niedrigerem Abstraktionsniveau realisiert werden:
  - Ping, TcpListener, TcpClient, UdpClient, NetworkStream, SslStream, ...
- Aber auch für Protokolle der Anwendungsschicht sind bereits Klassen vorhanden:
  - HttpClient, FtpWebRequest, SmtpClient, ...

# Sockets in C#

- Bekanntlich stellen die sog. **Sockets** (engl. für Steckverbindung) eine plattformunabhängige und standardisierte Programmierschnittstelle (API) für die Netzwerkprogrammierung dar.
  - Ein Programm kann einen Socket vom Betriebssystem anfordern und dieses dann zu Kommunikationszwecke einsetzen.
- Mit Hilfe von Sockets können Daten über das Netzwerk bzw. zwischen Programmen auf dem selben Rechner (Interprozesskommunikation) ausgetauscht werden.
  - Meist werden Sockets auf Basis der Kommunikationsprotokolle TCP bzw. UDP benötigt.
- Sockets werden in .Net meist nicht direkt verwendet.
  - Die Klassen `TcpClient` und `UdpClient` erzeugen automatisch entsprechende Sockets.

# Client-Server

- Wir wollen eine **Client-Server-Anwendung** schreiben.
  - Der Server soll die aktuelle Uhrzeit an alle verbundenen Clients verschicken.
- Entsprechend müssen wir zwei Projekt erstellen:
  - Eine Client-Anwendung und eine Server-Anwendung.
  - Beide Teile kommunizieren über das Netzwerk miteinander.
- Wir beginnen zunächst mit dem Client.




```
D:\Projekte\Chat\ChatClient\bin\Debug\C...
Connected...
11.12.2017 10:38:33
11.12.2017 10:38:34
11.12.2017 10:38:34
11.12.2017 10:38:35
11.12.2017 10:38:35
11.12.2017 10:38:36
11.12.2017 10:38:36
```

# TimeClient

- Wir erstellen eine Klasse TimeClient.
  - Dort nutzen wir die Klasse TcpClient, um eine Verbindung mit einem Server aufzubauen.
  - Der Klasse TcpClient muss eine IP-Adresse und ein Port übergeben werden.

```
public void ReadTime()
{
    using (var client = new TcpClient("127.0.0.1", 12345))
    {
        var task = ReadTimeAsync(client);
        Console.ReadLine();
    }
}
```



ReadTimeAsync  
erzeugt einen  
Task von  
ReadTime.

- Es ist **guter Stil**, die Verbindung in einem using-Block zu kapseln.
  - Beim Verlassen des Blocks wird die Verbindung wieder abgebaut.



# Zeit vom Server lesen

- Das Objekt vom Typ `TcpClient` repräsentiert den Datenkanal zum Server
  - Hier wird auch der eigentliche Socket verwaltet.
- Um Daten zu lesen und zu schreiben kann ein Stream-Objekt beschafft werden.

```
public void ReadTime(TcpClient client)
{
    using (var stream = client.GetStream())
    {
        while (true)
        {
            var s = ReadString(stream);
            Console.WriteLine(s);
        }
    }
}
```

Die Methode  
`ReadString`  
müssen wir noch  
erstellen.

# Streams lesen

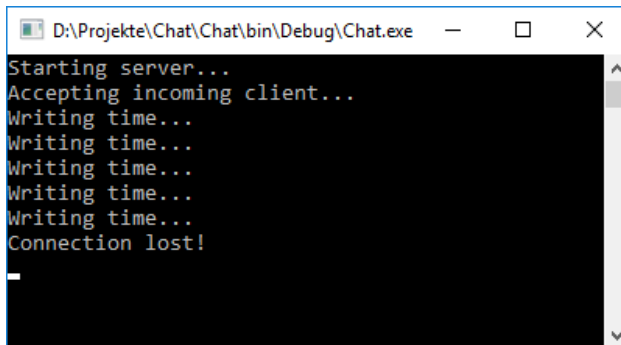
- Um Textnachrichten aus einem Netzwerk-Stream zu lesen, müssen die Rohdaten in ein string-Objekt umgewandelt werden.

```
private string ReadString(NetworkStream stream)
{
    var buffer = new byte[2048];
    int count = stream.Read(buffer, 0, buffer.Length);
    return Encoding.ASCII.GetString(buffer, 0, count);
}
```

- Dadurch haben wir nun alle Teile für den Zeit-Client beisammen.

# Zeitserver

- Als nächstes wird der Zeit-Server realisiert.
  - Dazu erstellen wir eine neue Klasse `TimeServer` mit der Methode `Listen`.
- Die Methode `Listen` nimmt neue Verbindungen auf Port 12345 entgegen.
  - Dazu wird die Klasse `TcpListener` benutzt.



```
D:\Projekte\Chat\Chat\bin\Debug\Chat.exe
Starting server...
Accepting incoming client...
Writing time...
Writing time...
Writing time...
Writing time...
Writing time...
Connection lost!
_
```

# Klasse TimeServer

- Beim Verbindungsaufbau eines Clients, liefert der TcpListener ein Objekt vom Typ TcpClient zurück.
  - Damit kann dann die Kommunikation mit dem Client abgewickelt werden.

```
public void Listen()
{
    var adress = IPAddress.Parse("127.0.0.1");
    var listener = new TcpListener(adress, 12345);
    Console.WriteLine("Starting server...");
    listener.Start();

    while (true)
    {
        var client = listener.AcceptTcpClient();
        WriteTimeAsync(client);
    }
}
```

Listener.Start() wartet auf eingehende Verbindungen.

WriteTimeAsync startet einen Task, um im Hintergrund Zeitangaben an den Client zu schicken.

# Daten schreiben

- Aus dem Client-Objekt kann wieder ein Stream ausgelesen werden.

```
using (var stream = client.GetStream())
{
    do
    {
        Console.WriteLine("Writing time...");

        if (!Write(stream, DateTime.Now.ToString()))
        {
            Console.WriteLine("Connection lost!");
            return;
        }

        Thread.Sleep(500);
    }
    while (true);
}
```

Die Methode  
Write müssen wir  
noch erstellen.

- Der Stream kann dann benutzt werden, um die Zeitangabe an den Client zu schicken.

# In Stream schreiben

- Um einen Text in einen Stream zu schreiben, muss dieser in ein Byte-Array umgewandelt werden.

```
private bool Write(NetworkStream stream, string text)
{
    var bytes = Encoding.ASCII.GetBytes(text);
    try
    {
        stream.Write(bytes, 0, bytes.Length);
    }
    catch
    {
        return false;
    }

    return true;
}
```

- Schlägt das Schreiben fehl, ist die Verbindung abgerissen.

# E-Mails versenden

- Die Klasse `TcpClient` kann auch dazu genutzt werden, um E-Mails zu versenden.
  - Wir müssen dazu eine Verbindung zu einem SMTP-Server aufbauen.
- Anschließend müssen wir uns an das SMTP-Protokoll halten.
  - Es müssen in einer genau definierten Reihenfolge bestimmte Nachrichten an den Server geschickt werden (Siehe nächste Folie).
- Die meisten SMTP-Server lassen heute nur eine verschlüsselte Verbindung zu.
  - Wir nutzen daher die Klasse `SslStream`, um den eigentlichen Stream darin zu verpacken.
  - Dadurch kann dann SSL-verschlüsselt kommuniziert werden.

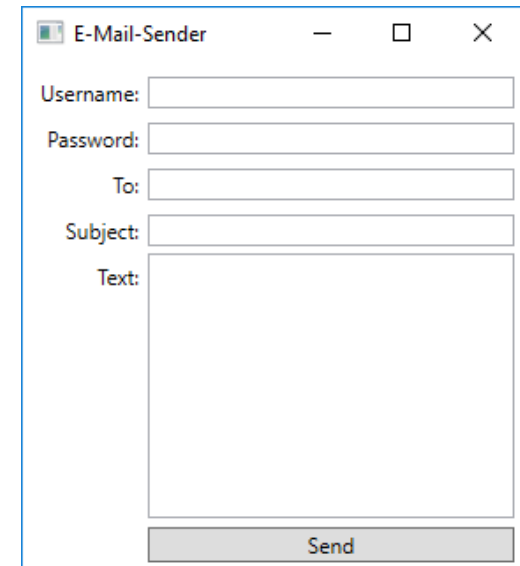
# Ablauf einer ESMTP Session

Server	Client
> 220 mail.example.org ESMTP	< EHLO example.net
> 250-example.org Hello example.net	
> 250 AUTH CRAM-MD5 LOGIN PLAIN	
> 334 VXNlcm5hbWU6	< AUTH LOGIN
> 334 UGFzc3dvcmQ6	< aGFucw==
> 235 ok	< c2Nobml0emVsbWl0a2FydG9mZmVsc2FsYXQ=
> 250 ok	< MAIL FROM:<hans@example.net>
> 250 ok	< RCPT TO:<fritz@example.org>
> 354 Go ahead.	< DATA
	< From:<hans@example.net>
	< To:<fritz@example.org>
	< Subject: Hallo
	<
	< Hallo Fritz.
> 250 Mail delivered.	< .
	< QUIT



# E-Mail-Client

- Um den E-Mail-Client zu realisieren, erstellen wir eine WPF-Anwendung.
  - Diese verfügt über einige Textfelder.
- Für den Verbindungsaufbau werden Benutzername und Passwort benötigt.
  - Die eigentliche E-Mail wird durch die Zieladresse, den Betreff und den Text der E-Mail abgefragt.
- Ein Button löst den eigentlichen Versand der E-Mail aus.



E-Mail-Sender

Username:

Password:

To:

Subject:

Text:

Send

# Versand der E-Mail

- Wir erstellen eine neue Klasse `SslSmtpMailer`
  - Im Konstruktor übergeben wir die Informationen, die zum Aufbau der Kommunikation notwendig sind.

```
var mailer = new SslSmtpMailer("smtp.googlemail.com", 465, username.Text,  
password.Password);  
var result = await mailer.SendMailAsync(to.Text, subject.Text, text.Text);
```

- Die Methode `SendMailAsync` versendet die übergebene E-Mail im Hintergrund.
  - Dazu erstellen wir einen Task, der die Methode `SendMail` asynchron aufruft.

```
public Task<bool> SendMailAsync(string to, string subject, string message)  
{  
    return Task.Factory.StartNew(() => SendMail(to, subject, message));  
}
```

# Versand der E-Mail

- In der Methode `SendMail` bauen wir zunächst die Verbindung auf.
  - Zudem wird ein SSL verschlüsselter Stream beschafft.

```
public bool SendMail(string to, string subject, string message)
{
    using (TcpClient mail = new TcpClient())
    {
        mail.Connect(host, port);

        using (SslStream sslStream = new SslStream(mail.GetStream()))
        {
            sslStream.AuthenticateAsClient(host);
```

- Anschließend werden Nachrichten gemäß dem SMTP-Protokoll an den Server verschickt und die korrekte Antwort geprüft, z.B.:

```
Write(sslStream, "MAIL FROM:<" + username + ">");
if (!Read(sslStream).Contains("OK"))
    return false;
```

# Web-Ressourcen

- Moderne Anwendungen greifen oft auf Ressourcen im World Wide Web zu.
  - Solche Ressourcen (z.B. Web-Seiten) werden von einem Server über das sog. **http-Protokoll (Hypertext Transfer Protocol)** angeboten.
  - Eine Ressource besitzt dabei eine eindeutige Adresse (URL).
- Das Http-Protokoll bietet unterschiedliche Anfragen an einen Server an:
  - Ein GET-Request ruft eine Ressource vom Server ab.
  - Ein POST-Request sendet Informationen an den Server, z.B. aus einem Formular.
  - Ein DELETE-Request löscht eine Ressource auf dem Server.
  - Ein PUT-Request ändert eine Ressource.

# HttpClient

- Theoretisch können wir einen Web-Client auch mit Hilfe der Klasse `TcpClient` umsetzen.
  - Es wird dann eine Verbindung zum Port 80 des Servers aufgebaut.
  - Die Anfrage muss dann das Http-Protokoll implementieren.
- Das .Net-Framework stellt für dafür allerdings eine eigene Klasse bereit: `HttpClient`.
  - Http-Requests werden dabei asynchron an den Server abgesetzt.
- Das Resultat ist immer vom Typ `Task<...>`.
  - Man benutzt folglich das Schlüsselwort `await`, um auf das Ergebnis zu warten.
- Ein GET-Request ist mit der Klasse sehr einfach umzusetzen:

```
var client = new HttpClient();  
var result = await client.GetAsync("http://www.google.de");  
var content = await result.Content.ReadAsStringAsync();
```



# Beispiel

- Der Speiseplan der Mensa Basilica in Hamm ist im Web abrufbar.
  - Wir können ein Programm schreiben, welches prüft, ob bestimmte Schlüsselwörter im Html-Quellcode der Web-Seite vorkommen.
  - z.B. Schnitzel, Pommes, Burger, ...
- Wir könnten dies auch mit unserem E-Mail-Programm kombinieren.
  - Es können E-Mails an eine Gruppe von Personen verschickt werden, wenn bestimmte Speisen angeboten werden.

# Speisenchecker

- Wir schreiben eine Methode, die eine Liste von Schlüsselwörtern übergeben bekommt.
  - Darin rufen wir die Web-Seite der Mensa ab.
  - Anschließend wird der Inhalt der Web-Seite geprüft.

```
public static async Task<bool> CheckFoodAsync(List<string> keywords)
{
    var client = new HttpClient();
    var result = await client.GetAsync(
        "http://www.studierendenwerk-pb.de/gastronomie/speiseplaene/mensa-basilica-hamm/");
    var content = await result.Content.ReadAsStringAsync();

    var text = content.ToUpper();
    foreach (var word in keywords)
        if (text.Contains(word.ToUpper()))
            return true;

    return false;
}
```

# Restful-Services

- Web-Seiten sind dazu gedacht, von Menschen in einem Browser betrachtet zu werden.
  - Im Gegensatz dazu bieten sog. **Restful-Services** zentralen Zugriff auf **strukturierte Daten** über das Http-Protokoll an.
- Dadurch können verteilte Anwendungen realisiert werden.
  - Ein Client lädt Daten von einem Server nach.
  - Diese Daten sind in einem maschinenlesbaren Format, z.B. JSON, abgelegt.
  - z.B. kann der Zugriff auf eine Produkt- oder Kundendatenbank angeboten werden.
- Web-Adressen werden bei Restful-Services nach einem bestimmten System vergeben:
  - Eine Liste von Kunden könnte z.B. über die URL <http://data.com/customers> per GET-Request abgerufen werden.
  - Ein spezifischer Kunde mit der Kundennummer 17 kann über die URL <http://data.com/customers/17> abgefragt, geändert oder gelöscht werden.



# JSON-Daten

- Bei Restful-Services werden die Daten häufig im sog. **JSON-Format (abgekürzt für Javascript Object Notation)** ausgetauscht.
  - Dadurch werden die Daten für Programm lesbar.
- JSON ist ähnlich (aber schlanker) als XML.
  - Daten werden in Form von Schlüssel-Wert-Paaren dargestellt.
  - Mehrere Daten werden durch Komma getrennt.
  - Geschweifte Klammern repräsentieren Objekte.
  - Eckige Klammern repräsentieren Arrays.
- Beispiel:
  - Der folgende JSON-Ausschnitt repräsentiert eine Liste von Person-Objekten:

```
[  
  { "name": "John", "age": 30 },  
  { "name": "Peter", "age": 21 }  
]
```

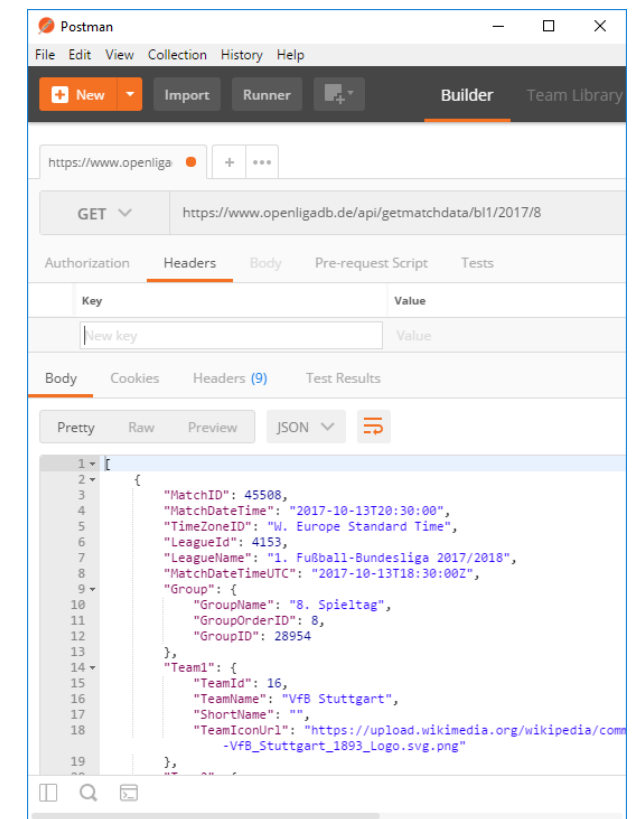
# JSON-Daten verarbeiten

- Für die Verarbeitung von JSON-Daten stehen bereits fertige NuGet-Pakete bereit.
  - Ein sehr beliebtes Paket ist **Newtonsoft.JSON**.
  - Es kann mit dem Befehl `Install-Package Newtonsoft.Json` im Paketmanager installiert werden.
- Newtonsoft.JSON bietet zwei Wege an, um JSON-Daten zu verarbeiten.
  - Das manuelle Parsen der Daten mit Hilfe der Klasse `JArray` und `JObject`.
  - Das sog. Deserialisieren von JSON in passende Objekte.

# Beispiel OpenLigaDB



- Die OpenLigaDB ist ein Restful-Service, bei dem die Spieldaten der Fußball-Bundesliga abgerufen werden können.
  - Der 8. Spieltag in der Saison 2017/2018 kann z.B. über die Adresse <https://www.openligadb.de/api/getmatchdata/bl1/2017/8> abgerufen werden.
- Wir können solche Services z.B. mit einem Programm namens **Postman** testen.
  - Damit können unterschiedlichen Anfragen an Restful-Services gestellt und die Antwort untersucht werden.
- Im Folgenden wollen wir ein Programm schreiben, welches die Daten aus dem Service lädt.
  - Anschließend sollen die JSON-Daten ausgewertet werden.



# Daten manuell parsen

- Die Daten laden wir mit Hilfe eines GET-Request:

```
var client = new HttpClient();  
var result = await client.GetAsync("https://www.openligadb.de/api/...");  
var content = await result.Content.ReadAsStringAsync();
```

- Nun benutzen wir die Klassen JObject und JArray aus NuGet-Paket.
  - Dadurch können wir auf die einzelnen Bestandteile zugreifen.
  - In diesem Fall geben wir die Namen der an den beteiligten Spielen auf der Konsole aus:

```
var jarray = JArray.Parse(content);  
foreach (var obj in jarray) {  
    var team1 = obj["Team1"]["TeamName"];  
    var team2 = obj["Team2"]["TeamName"];  
    Console.WriteLine(team1 + " vs " + team2);  
}
```



```
stuckenholz - Externe Visual Studio-Konsole - mono32 + bas...  
VfB Stuttgart vs 1. FC Köln  
1. FSV Mainz 05 vs Hamburger SV  
Hannover 96 vs Eintracht Frankfurt  
Bayern München vs SC Freiburg  
TSG 1899 Hoffenheim vs FC Augsburg  
Hertha BSC vs FC Schalke 04  
Borussia Dortmund vs RB Leipzig  
Bayer 04 Leverkusen vs VfL Wolfsburg  
Werder Bremen vs Borussia Mönchengladbach
```

# Daten deserialisieren

- Der Zugriff auf die JSON-Daten kann aber noch eleganter umgesetzt werden.
  - Dazu müssen Klassen genau so erstellt werden, dass sie zu der Struktur der JSON-Daten passen.
- Wir wollen Spiel-Daten aus der OpenLigaDB abrufen
  - Entsprechend müssen wir die Klassen Match, Group, Team usw. mit passenden Eigenschaften erstellen.
- Danach können mit Hilfe von Newtonsoft.JSON die JSON-Daten in Objekte dieser Klassen umgewandelt werden.
  - Ein solcher Vorgang wird auch als Deserialisieren bezeichnet.

# JSON und Klassen

```

1  [
2  {
3      "MatchID": 45508,
4      "MatchDateTime": "2017-10-13T20:30:00",
5      "TimeZoneID": "W. Europe Standard Time",
6      "LeagueId": 4153,
7      "LeagueName": "1. Fußball-Bundesliga 2017/2018",
8      "MatchDateTimeUTC": "2017-10-13T18:30:00Z",
9      "Group": {
10         "GroupName": "8. Spieltag",
11         "GroupOrderID": 8,
12         "GroupID": 28954
13     },
14     "Team1": {
15         "TeamId": 16,
16         "TeamName": "VfB Stuttgart",
17         "ShortName": "",
18         "TeamIconUrl": "https://upload.wikimedia.org/
19     },
20     "Team2": {
21         "TeamId": 65,
22         "TeamName": "1. FC Köln",
23         "ShortName": "",
24         "TeamIconUrl": "https://www.openligadb.de/ima
25     },
26     "LastUpdateDateTime": "2017-10-17T18:14:11.377",

```



```

public class Match
{
    public Match()
    {
        MatchResults = new List<MatchResult>();
    }

    public Team Team1 { get; set; }
    public Team Team2 { get; set; }
    public List<MatchResult> MatchResults { get; set; }
    public DateTime MatchDateTime { get; set; }
    public int MatchID { get; set; }
    public bool MatchIsFinished { get; set; }
    public Group Group { get; set; }
}

```

# Daten verarbeiten

- Auf diese Weise können wir die Spiel-Daten leicht laden und in eine Liste von Match-Objekten deserialisieren:

```
public static async Task<List<Match>> LoadMatches()
{
    var client = new HttpClient();
    var result = await client.GetStringAsync("https://www.openligadb.de/api/getmatchdata/b11/2017");

    JArray array = JArray.Parse(result);
    return array.ToObject<List<Match>>();
}
```

- Mit einer solchen Liste von Spielen können nun diverse Auswertungen erfolgen:

```
public static int GetAverageHomeGoals(List<Match> matches)
{
    int sum_home_goals = 0;

    foreach (var m in matches)
        if (m.MatchIsFinished)
            sum_home_goals += m.FinalResult.PointsTeam1;

    return sum_home_goals;
}
```

# Weitere Restful-Services

- Es existieren viele kostenlos nutzbare Restful-Services.
  - Damit könne unterschiedlichste Daten abgerufen werden.
  - Eine Liste von solchen Diensten kann z.B. hier gefunden werden: <https://github.com/toddmotto/public-apis>
  - Oder hier: <https://www.programmableweb.com/>
- Dort finden sich sehr nützliche Dienste:
  - IEX → Abrufen von Kursen von Aktien.
  - Yahoo! Weather → Wetterdaten.
  - Gmail → Zugang zum eigenen Mail-Account.
  - ...



# Wir haben heute gelernt...

- Was Sockets sind.
- Was Client-Server-Kommunikation bedeutet.
- Wie man die Klasse TcpClient einsetzt.
- Wie man einen Zeitclient und -Server aufbaut.
- Wie man E-Mails über SMTP versenden kann.
- Wie Web-Ressourcen mit dem HttpClient abgerufen werden.
- Die Implementierung des Speisencheckers.
- Was Restful-Services sind.
- Wie man JSON-Daten in C# verarbeiten kann.
- Wie man Fussball-Daten von der OpenLigaDB abrufen.