Aufgabe

Als Aufgabe steht diesmal die Wahl eines Tools für das Buildmanagement an. Mit diesem Tool soll sich näher beschäftigt werden sowie ein Buildscript für die automatisierte und wiederholbare Ausführung eines Builds erstellt werden.

Auswahl des Tools

Als Buildmanagement Tool wähle ich **NUKE**[1] als eines der wenigen Tools, welche speziell in Richtung **C#** und dem **.NET Framework** zielen.

Hierbei bietet das Tool eine Reihe von Vorteilen:

- Die erstellten Build Skripte werden als **eigenständige C# Konsolen Applikationen** erstellt und werden Teil der Solution. Es besteht keine Notwendigkeit, eine neue Sprache zu lernen.
- Durch die **Ähnlichkeit zu bekannten System wie ANT** ist es relative innovativ (zumindest habe ich mich ohne Vorkenntnisse in Buildmanagement Tool schnell einfinden können).
- Hierbei stellt **NUKE** eine eigene **C# DSL** zur Verfügung sowie ein eigenes Ökosystem. Zudem unterstützt es mit eigenen Routinen zum Implementierung des Build Systems in Projekten.
- Die Erstellung und Test der Build Scripte erfolgt direkt **innerhalb der IDE**. Hierbei werden neben Visual Studio auch JetBrains Rider und VS Code unterstützt. Ebenso existieren eigene Plugins für die Umgebungen.
- Durch die Erstellung innerhalb einer IDE erfolgt die **volle Sprachunterstützung** in Form von Intellisense oder der Möglichkeit zum Debugging. Zudem werden die mächtigen Funktionen der IDE und der .NET CLI zur Erstellung von Builds genutzt. Task können so extrem einfach gehalten werden und werden durch Intellisense unterstützt.
- **Build Scripte** sind somit **im Team bearbeitbar** und mitsamt des Projektes beispielsweise **auf GIT verwaltbar**. Alle Vorteile einer Versionsverwaltung gelten somit auch für das Buildsystem.
- Nuke ist **OpenSource** und auf **GitHub** frei verfügbar und baut auf einer **erweiterbaren Architektur** auf. Das eröffnet die Möglichkeit, es theoretisch für alle möglichen Belange zu verwenden und anzupassen.
- Zur Nutzung reicht ist die einmalige Installation des GlobalTools sinnvoll, um ein automatisiertes Setup ausführen zu können. Dies erspart Zeit und erspart das manuelle Anlegen. Um eine Konsolenapplikation zu einem **NUKE** Build Projekt zu machen, reicht das Hinzufügen eines **einzelnen NuGet Packages**.

[1] https://www.nuke.build

- Nuke unterstützt **.NET Core 5**, **.NET Framework** und **Mono** und kann so unter **Windows**, **Linux** und **Apple** verwendet werden.
- Nuke ist für die Anbindung an **CLI Tools** und **CI Systemen** vorbereitet, hier namentlich u.a. für **Git Action**, **Azure Pipelines** und **JetBrains TeamCity**. Ebenso sind die Durchführung von **Tests** und das Einbinden von beispielsweise **Metriken** relativ einfach möglich.

Trotz einer recht umfangreichen Dokumentation und API Referenz stellt die Unterstützung unter Umständen ein Problem dar, da diese eher auf erfahrene Anwender abzielt, nicht auf Anfänger. Auch findet man im Internet im Vergleich zu anderen Themen eher wenige Informationen und muss länger suchen. Trotz dessen lassen sich Probleme in der Regel lösen.

Verwendete Beispielprojekt

Als Beispielprojekt zur Demonstration des Tools verwende ich das bereits aus den vorherigen Aufgaben bekannte Spiel **TicTacToe**. Um es von den vorherigen Abgaben zu separieren habe ich es in ein neues Repository auf **GIT**[2] ausgelagert.

Sämtliche hier beschriebenen Arbeiten, darunter auch das Script, können dort direkt im Code eingesehen werden. Somit stellt es ein komplettes Beispiel zur Nutzung von **NUKE** in einen überschaubaren Projekt dar.

Im folgenden gehe ich auf das Programm selbst nicht näher ein, da es nicht im Fokus dieser Arbeit liegt.

Angestrebte Ergebnis

Es soll für das Programm ein automatisierter Build erstellt werden, der unter Windows, Linux und Apple ausgeführt werden kann. Dieser Build soll aus einer Kommandozeile heraus ohne aktive geöffneten Projekt ausgeführt werden können. Folgende Aufgaben sollen hierbei erledigt werden:

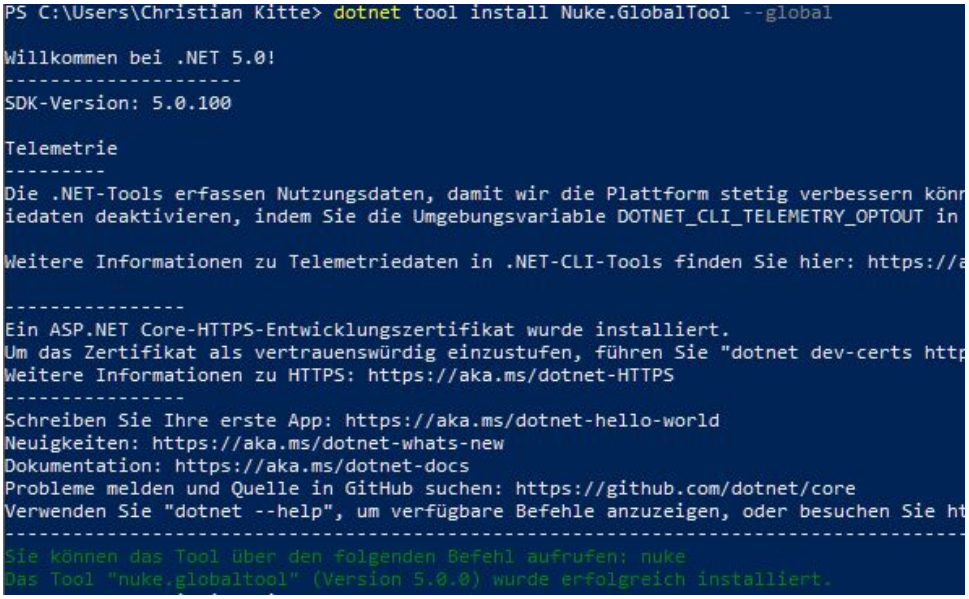
- Erstellen des Builds als Release ohne Rücksicht auf aktuelle Projekteinstellungen
- Kopieren aller neu erstellten Kompilate in ein explizit geleertes Verzeichnis
- Kopieren eines “Tutorials” zur Demonstration des Hinzufügens weiterer Artefakte

Es wird davon ausgegangen, dass es sich bei dem Zielsystem um einen Rechner mit aktueller **Visual Studio** Installation sowie **.NET Core 5** handelt. Hierdurch sind auch die **dotnet Tools** verfügbar.

[2] https://github.com/ChristianKitte/TicTocToeBuildMgm

Vorbereitung für den Einsatz von NUKE

Auch wenn für den Einsatz von **NUKE** grundsätzlich nur ein **NuGet Package** notwendig ist, wird der Einsatz des **NUKE Setup** empfohlen. Die Installation kann direkt über die **Powershell** mit Hilfe der **dotnet Tools** erfolgen, wie in der unten stehenden Abbildung zu sehen ist:



Ab diesem Zeitpunkt ist es möglich, für eine beliebige Solution mit Hilfe des Befehls **NUKE :Setup** ein **Konsolenprojekt** erstellen zu lassen und es der jeweiligen **Projektmappe** hinzuzufügen.

Hinzufügen eines Buildprojektes zum Projekt TicTocToe

In einem ersten Schritt habe ich zunächst einen Klon aus meinem neuen **GIT Repository**[3] lokal rüber kopiert. Anschließend kann in das entsprechende Projektverzeichnis gewechselt werden. Es existieren die unten zu sehenden Dateien, die die Solution darstellen.

[3] https://github.com/ChristianKitte/TicTocToeBuildMgm

```
PS C:\Users\Christian Kitte> cd "C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm"
PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm> dir


    Verzeichnis: C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        04.12.2020     18:58                .vs
d-----        04.12.2020     18:58                TicTocLib
d-----        04.12.2020     18:58                TicTocToe
-a----        04.12.2020     18:58           2581 .gitattributes
-a----        04.12.2020     18:58           6084 .gitignore
-a----        04.12.2020     18:58         392472 CCD Cheat Sheet.pdf
-a----        04.12.2020     18:58              0 CCD Codeänderungen.txt
-a----        04.12.2020     18:58           6011 README.md
-a----        04.12.2020     18:58           1871 TicTocToe.sln


PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm>
```

Im nächsten Schritt wird nun die Unterstützung für **NUKE** für die Solution hinzugefügt. Dies kann manuell geschehen, einfacher ist jedoch die Unterstützung mit der **Setup Routine** von **NUKE** wie im folgenden zu sehen ist:

```
PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm> nuke :setup
NUKE Global Tool version 5.0.0 (Windows,.NETCoreApp,Version=v2.1)
How should the build project be named?
→ _build
Where should the build project be located?
→ ./build
Which NUKE version should be used?
→ 5.0.0 (latest release)
Which solution should be the default?
→ TicTocToe.sln
Do you need help getting started with a basic build?
→ Yes, get me started!
Restore, compile, pack using ...
→ dotnet CLI
Source files are located in ...
→ ./src
Move packages to ...
→ ./output
Where do test projects go?
→ ./tests
Do you use GitVersion?
→ No, custom versioning
Creating directory 'C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm\.\build'...
Creating directory 'C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm\src'...
Creating directory 'C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm\tests'...
PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm>
```

Im Verlauf der Installation werden interaktiv **verschiedene Parameter** abgefragt, hierunter der **Name** der neu zu erstellenden Konsolenapplikation, ein **Default Projekt** und eventuelle **Unterstützung für GIT**.

Die aktuell gemachten Einstellungen sind in Grün zu sehen. Besonders interessant ist die Einstellung “**Yes, get me started!**”, welche ein **Basic Script** zur Verfügung stellt und so Arbeit spart.

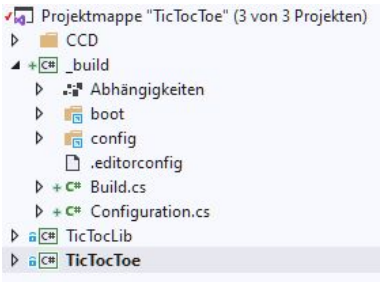
Die Setup Routine fügt der Solution eine neue **Konsolenapplikation** hinzu und fügt dieser das **NUKE NuGet Package** in der aktuellsten Version hinzu. Zudem werden alle benötigten **Abhängigkeiten** eingetragen und **Konfigurationsdateien** erstellt.

Das so erstellte **Grundgerüst** ist bereits grundsätzlich **voll einsatzfähig** und **kann ausgeführt werden**. Auf Dateiebene ergibt sich das folgende Bild:

```
PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm> dir


    Verzeichnis: C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        04.12.2020     19:03                .vs
d-----        04.12.2020     19:03                build
d-----        04.12.2020     19:03                src
d-----        04.12.2020     19:03                tests
d-----        04.12.2020     18:58                TicTocLib
d-----        04.12.2020     18:58                TicTocToe
-a----        04.12.2020     18:58           2581 .gitattributes
-a----        04.12.2020     18:58           6084 .gitignore
-a----        04.12.2020     19:03             13 .nuke
-a----        04.12.2020     19:03            207 build.cmd
-a----        04.12.2020     19:03           2999 build.ps1
-a----        04.12.2020     19:03           2336 build.sh
-a----        04.12.2020     18:58         392472 CCD Cheat Sheet.pdf
-a----        04.12.2020     18:58              0 CCD Codeänderungen.txt
-a----        04.12.2020     18:58           6011 README.md
-a----        04.12.2020     19:03           2178 TicTocToe.sln


PS C:\Users\Christian Kitte\Source\Repos\TicTocToeBuildMgm>
```

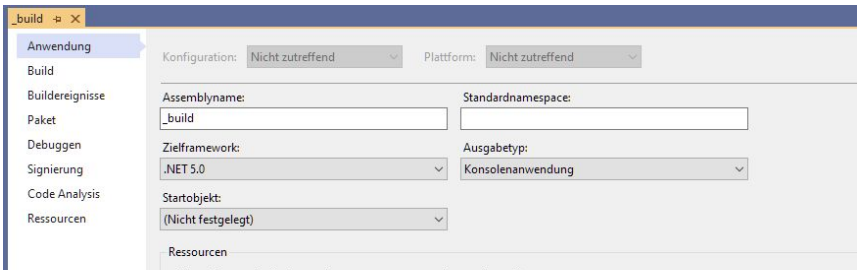
Sicht aus der IDE Perspektive

Wenn nunmehr das Projekt in Visual Studio geöffnet wird, ergibt sich im **Projektfenster** die folgende Ansicht. Sollte Visual Studio bereits geöffnet gewesen sein, so wird lediglich das Projekt neu geladen.



Ein Punkt, der nicht sofort gesehen wird ist, dass das generierte Projekt noch manuell auf **.NET Core 5**[4] eingestellt werden muss. Hier sagt jedoch die Dokumentation, dass dies nicht zwingend für den Build auch eines Core 5 Projektes notwendig ist.

[4] im Build ist bereits die neue Sprechweise mit .NET 5.0 als Target eingetragen



Die Build.cs

Alle Dateien im Detail zu betrachten, würde bei weitem zu umfangreich für diese Aufgabe werden. Stattdessen sollen nur zwei, drei wichtige Dateien betrachtet werden.

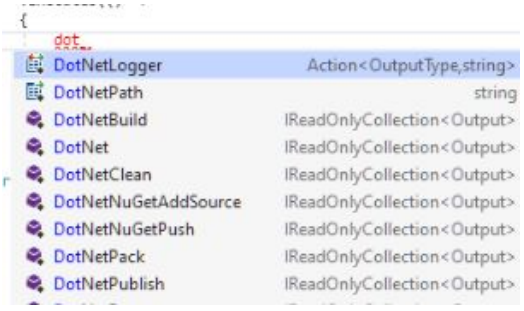
Kernstück der neuen Dateien ist die Datei **Build.cs**. Sie leitet sich von der Klasse **NukeBuild** ab und bietet umfangreiche Funktionalitäten zur Steuerung des Builds.

In der ganz unten zu sehenden kompletten Codebasis erkennt man, dass bereits drei Task definiert sind: **Compile**, **Restore** und **Clean**. Während die letzten beiden lediglich der Vorbereitung eines neuen Builds dienen, führt Compile den eigentliche Build aus.

Hierfür nutzt er die Klasse **DotNetBuild**, welche diverse Hilfsfunktionen zum Builden bietet. Interessant ist dessen Einsatz und Gebrauch als **interne DSL**:

```
DotNetBuild( configurator: s :DotNetBuildSettings => s
    .SetProjectFile(Solution)
    .SetConfiguration(Configuration)
    .EnableNoRestore());
```

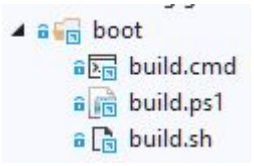
Neben **DotNetBuild** existieren weiteren Hilfsklassen mit Funktionalitäten für verschiedene Domänen:



So liefert **DotNetPack** diverse Routinen zur Erstellung von **NuGet Packages**. Das folgende Bild zeigt die von der Setup Routine erstellte **Build.cs** komplett:

```
[CheckBuildProjectConfigurations]
[ShutdownDotNetAfterServerBuild]
class Build : NukeBuild
{
    /// Support plugins are available for:
    ///   - JetBrains ReSharper        https://nuke.build/resharper
    ///   - JetBrains Rider            https://nuke.build/rider
    ///   - Microsoft VisualStudio     https://nuke.build/visualstudio
    ///   - Microsoft VSCode           https://nuke.build/vscode

    public static int Main () => Execute<Build>(params defaultTargetExpressions: x :Build => x.Compile);

    [Parameter(description: "Configuration to build - Default is 'Debug' (local) or 'Release' (server)")]
    readonly Configuration Configuration = IsLocalBuild ? Configuration.Debug : Configuration.Release;

    [Solution] readonly Solution Solution;
    [GitRepository] readonly GitRepository GitRepository;

    AbsolutePath SourceDirectory => RootDirectory / "src";
    AbsolutePath TestsDirectory => RootDirectory / "tests";
    AbsolutePath OutputDirectory => RootDirectory / "output";

    Target Clean => _ => _
        .Before(Restore)
        .Executes(() =>
        {
            SourceDirectory.GlobDirectories(params patterns: "**/bin", "**/obj").ForEach(DeleteDirectory);
            TestsDirectory.GlobDirectories(params patterns: "**/bin", "**/obj").ForEach(DeleteDirectory);
            EnsureCleanDirectory(OutputDirectory);
        });

    Target Restore => _ => _
        .Executes(() =>
        {
            DotNetRestore(configurator: s :DotNetRestoreSettings => s
                .SetProjectFile(Solution));
        });

    Target Compile => _ => _
        .DependsOn(Restore)
        .Executes(() =>
        {
            DotNetBuild(configurator: s :DotNetBuildSettings => s
                .SetProjectFile(Solution)
                .SetConfiguration(Configuration)
                .EnableNoRestore());
        });
}
```

Wichtig sind die hier zu sehenden **Task'**, welche quasi als **auszuführende Blöcke** fungieren. Hierbei können von der Intention her zwei Bereiche unterschieden werden. Der erste Bereich macht Vorgaben über Abhängigkeiten wie beispielsweise **.DependsOn** oder Reihenfolgen wie beispielsweise **.Before**. Der zweite Bereich wird durch **.Executes** eingeleitet und definiert den eigentlich auszuführenden Bereich.



Im Verzeichnis **boot** befinden sich darüber hinaus drei weitere Dateien, welche zur Ausführung der Konsolenapplikation genutzt werden können.

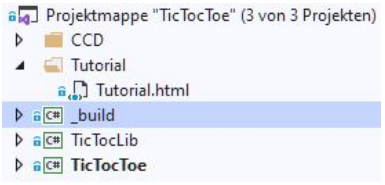
Hierbei handelt es sich bei der Datei **build.ps1** um ein Powershell Skript und bei der Datei **build.sh** für eine unter Linux ausführbare Datei.

Eine weitere Option ist jedoch die Ausführung direkt innerhalb der Shell, wie sie am Schluß vorgestellt wird. Zusätzlich findet sich im Ordner **config** die Datei **.nuke**, welche aktuell nur die Default Solution enthält.

Grundsätzlich ist bei der Bearbeitung und der Entdeckung der Dateien gerade auch für Anfänger die sehr umfangreiche Dokumentation und Unterstützung durch die IDE sehr nützlich und sinnvoll:

Anpassen der Build.cs

In diesem Abschnitt soll kurz auf die Anpassung der Build auf die oben genannten Anforderungen eingegangen werden. Hierbei ist an dem zuvor generierten Gerüst im Grunde nichts mehr zu machen.



Zu Anfang gehen ich nochmals auf die Situation im Projekt ein. Es existiert eine Konsolen Applikation **_build**, eine Library **TicTocLib** sowie das eigentliche Programm **TicTocToe**. Im Verzeichnis Tutorial befindet sich eine HTML Datei **Tutorial.html**, welche die Dokumentation darstellt.

Im ersten Schritt ergänze ich die Klasse um weitere Pfadangaben zum **tutorial** sowie den Pfad zur Ausgabe des Kompilats (**net5.0**). Der Ordner **output** dient als Sammelordner für alle Artefakte:

```
AbsolutePath OutputDirectory => RootDirectory / "output";
AbsolutePath TutorialDirectory => RootDirectory / "tutorial";
AbsolutePath BinaryDirectory => RootDirectory / "TicTocToe" / "bin" / "Release" / "net5.0";
```

Hierbei könnte man auch dynamisch auf die in der Solution definierten Projekte zugreifen, was ich aus Gründen der Übersichtlichkeit nicht gemacht habe.

Als nächsten Schritt erstelle ich einen neuen Task, den ich **Deploy** nenne. Aufgabe des Task ist es, alle Ausgaben in das Verzeichnis **output** zu kopieren.

```
Target Deploy => _ => _
    .DependsOn(Compile)
    .Requires(() => System.IO.File.Exists(path: TutorialDirectory + "\\tutorial.html"))
    .Executes(() =>
    {
        System.IO.File.Copy(sourceFileName: TutorialDirectory + "\\tutorial.html", destFileName: OutputDirectory + "\\tutorial.html");

        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.exe", destFileName: OutputDirectory + "\\TicTocToe.exe");
        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.dll", destFileName: OutputDirectory + "\\TicTocToe.dll");
        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.dll.config", destFileName: OutputDirectory + "\\TicTocToe.dll.config");
        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.runtimeconfig.json",
            destFileName: OutputDirectory + "\\TicTocToe.runtimeconfig.json");
        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocLib.dll", destFileName: OutputDirectory + "\\TicTocLib.dll");
        System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\Autofac.dll", destFileName: OutputDirectory + "\\Autofac.dll");
    });
```

Wie zu erkennen ist, handelt es sich um normale C# Befehle, welche mit Hilfe der **Copy** Methode die Kopiervorgänge ausführt.

Die Auslagerung in einem eigenen Task ermöglicht es zudem, **nur** einen Kopiervorgang anzustoßen durch expliziten Ausführung von nur diesen Task'. Hierzu am Ende mehr.

Den Task **Deploy** stelle ich innerhalb der Methode **Main** als Default Task ein.

```
public static int Main() => Execute<Build>(params defaultTargetExpressions: x:Build => x.Deploy);
```

Interessant ist die Option **.Requires()**. Letzteres wird eine **Action** übergeben, welche somit Prüfung frei programmierbar macht.

Im oben zu sehenden Code prüft sie lediglich die Verfügbarkeit des Datei **Tutorial.html**. Eine Änderung dieser Datei führt zu der Ausgabe:

```
Exception: Assertion failed: Target 'Deploy' requires 'Exists((Convert(value(Build).
TutorialDirectory, String) + "\tutorial.html"))'.
   at Nuke.Common.ControlFlow.Fail(String text)
   at Nuke.Common.Execution.RequirementService.ValidateRequirements(NukeBuild build,
 IReadOnlyCollection`1 executingTargets)
   at Nuke.Common.Execution.BuildExecutor.Execute(NukeBuild build, IReadOnlyCollecti
on`1 skippedTargets)
   at Nuke.Common.Execution.BuildManager.Execute[T](Expression`1[] defaultTargetExpr
essions)

Repeating warnings and errors:
Exception: Assertion failed: Target 'Deploy' requires 'Exists((Convert(value(Build).
TutorialDirectory, String) + "\tutorial.html"))'.

Target                Status        Duration
Clean                 NotRun            0:00
Restore               NotRun            0:00
Compile               NotRun            0:00
Deploy                NotRun            0:00
Total                                   0:00

Build failed on 06.12.2020 17:26:32. (?°?°) ??  ???
```

Die **Compile** Methode braucht hingegen nicht geändert zu werden, da hier nur die Default Solution kompiliert wird. Ebenso verhält es sich mit **Restore** und **Clean**.

Somit ist selbst bei nur grundsätzlicher Kenntnis von NUKE viel mit relativ wenige Aufwand und Anpassungen zu erreichen.

Zusammenspiel der Task

Gerade für diejenigen, welche sich zuvor noch nicht aktiv mit Build Tools beschäftigt haben, gebraucht es etwas Übung, bis man das Zusammenspiel auch praktisch grundsätzlich im Griff hat.

NUKE arbeitet grundsätzlich (ähnlich wie **ANT**) mit Task', welche als Ganzes orchestriert werden müssen. Der wichtigste Befehl hierfür ist **.DependsOn(X)**, wobei X ein oder mehr **zuvor(!)** auszuführende Task' definiert.

Zusätzlich kann jeder Task über **.Before(X)** bzw. **.After(X)** gesteuert werden, um bei Gleichzeitigkeit trotzdessen eine Reihenfolge zu erzwingen.

Interessant ist auch die Option **.TriggeredBy(X)** sowie die oben bereits erwähnte **.Requires()**. Letzteres wird eine Action übergeben, welche somit Prüfung frei programmierbar macht.

Zusätzlich existieren sogenannte **Dynamic** und **Static Conditions**, welche genau vor der Ausführung evaluiert werden. Mit **Failure Behavior**, was eine Art Try-Catch Mechanismus darstellt, können Probleme abgefangen werden. Abgeschlossen werden die Funktionalitäten durch die von den Task ausgelösten **Events** wie **OnBuildCreate()**. Diese sind C# konform implementiert.

Viele der zuvor genannten Optionen existieren auch als eine Version mit Fehlerabfangung. Diese haben den Präfix wie beispielsweise **.TryBefore(X)**. Für die hier gesuchte Lösung habe ich die Task' wie folgt orchestriert:

```
Target Clean => _ => _
    .Before()
    .After()
    .Executes(() =>
    {
    });

Target Restore => _ => _
    .DependsOn(Clean)
    .Executes(() =>
    {
    });

Target Compile => _ => _
    .DependsOn(Restore)
    .Executes(() =>
    {
    });

Target Deploy => _ => _
    .DependsOn(Compile)
    .Requires(() => System.IO.File.Exists(path: TutorialDirectory + "\\tutorial.html"))
    .Executes(() =>
    {
    });
```

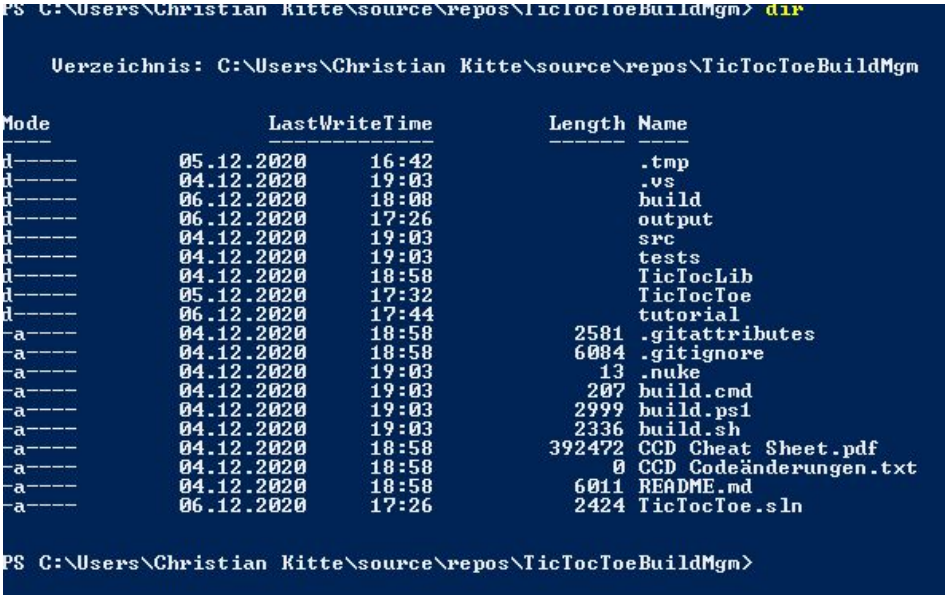
Deploy basiert auf **Compile**, **Compile** auf **Restore** und **Restore** auf **Clean**. Wird **Deploy** nun zur Ausführung gebracht, führt dies dazu, dass zunächst **Clean**, dann **Restore**, dann **Compile** und zuletzt **Deploy** zur Ausführung kommt. Dies natürlich nur, insofern die jeweils benötigten Task erfolgreich zur Ausführung kamen.

Im folgenden sieht man nochmal den gesamten angepassten Code der Build.cs:

```
using System;
using System.Linq;
using Nuke.Common;
using Nuke.Common.CI;
using Nuke.Common.Execution;
using Nuke.Common.Git;
using Nuke.Common.IO;
using Nuke.Common.ProjectModel;
using Nuke.Common.Tooling;
using Nuke.Common.Tools.DotNet;
using Nuke.Common.Utilities.Collections;
using static Nuke.Common.EnvironmentInfo;
using static Nuke.Common.IO.FileSystemTasks;
using static Nuke.Common.IO.PathConstruction;
using static Nuke.Common.Tools.DotNet.DotNetTasks;

[CheckBuildProjectConfigurations]
[ShutdownDotNetAfterServerBuild]
class Build : NukeBuild
{
    /// Support plugins are available for:
    ///   - JetBrains ReSharper        https://nuke.build/resharper
    ///   - JetBrains Rider            https://nuke.build/rider
    ///   - Microsoft VisualStudio     https://nuke.build/visualstudio
    ///   - Microsoft VSCode           https://nuke.build/vscode

    public static int Main() => Execute<Build>(params defaultTargetExpressions: x => x.Deploy);

    [Parameter(description: "Configuration to build - Default is 'Debug' (local) or 'Release' (server)")]
    readonly Configuration Configuration = IsLocalBuild ? Configuration.Debug : Configuration.Release;

    [Solution] readonly Solution Solution;

    [GitRepository] readonly GitRepository GitRepository;

    AbsolutePath SourceDirectory => RootDirectory / "src";
    AbsolutePath TestsDirectory => RootDirectory / "tests";
    AbsolutePath OutputDirectory => RootDirectory / "output";
    AbsolutePath TutorialDirectory => RootDirectory / "tutorial";
    AbsolutePath BinaryDirectory => RootDirectory / "TicTocToe" / "bin" / "Release" / "net5.0";

    Target Clean => _ => _
        .Executes(() =>
        {
            SourceDirectory.GlobDirectories(params patterns: "**/bin", "**/obj").ForEach(DeleteDirectory);
            TestsDirectory.GlobDirectories(params patterns: "**/bin", "**/obj").ForEach(DeleteDirectory);

            EnsureCleanDirectory(OutputDirectory);
        });

    Target Restore => _ => _
        .DependsOn(Clean)
        .Executes(() =>
        {
            DotNetRestore(configurator: s => s
                .SetProjectFile(Solution));
        });

    Target Compile => _ => _
        .DependsOn(Restore)
        .Executes(() =>
        {
            DotNetBuild(configurator: s => s
                .SetProjectFile(Solution)
                .SetConfiguration(Configuration)
                .EnableNoRestore());
        });

    Target Deploy => _ => _
        .DependsOn(Compile)
        .Requires(() => System.IO.File.Exists(path: TutorialDirectory + "\\tutorial.html"))
        .Executes(() =>
        {
            System.IO.File.Copy(sourceFileName: TutorialDirectory + "\\tutorial.html", destFileName: OutputDirectory + "\\tutorial.html");

            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.exe", destFileName: OutputDirectory + "\\TicTocToe.exe");
            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.dll", destFileName: OutputDirectory + "\\TicTocToe.dll");
            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.dll.config", destFileName: OutputDirectory + "\\TicTocToe.dll.config");
            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocToe.runtimeconfig.json",
                destFileName: OutputDirectory + "\\TicTocToe.runtimeconfig.json");
            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\TicTocLib.dll", destFileName: OutputDirectory + "\\TicTocLib.dll");
            System.IO.File.Copy(sourceFileName: BinaryDirectory + "\\Autofac.dll", destFileName: OutputDirectory + "\\Autofac.dll");
        });
}
```

Ausführung des Buildskriptes

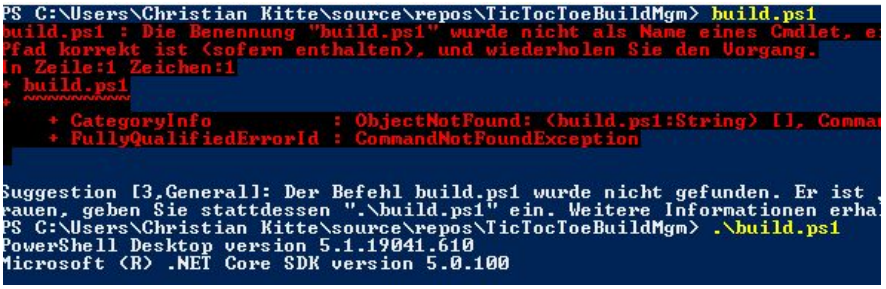
Zur Ausführung soll hier nur die mithilfe der **Powershell** vorgestellt werden. Hierfür wechselt man in das Verzeichnis der Solution.



Man erkennt, dass die Dateien **build.cmd**, **build.ps1** sowie **build.sh** hier liegen. Somit wird klar, dass sie nur innerhalb der IDE in einem **logischen Ordner boot** existieren.

Ausführung als Powershell Skript

Bei der Ausführung des **Powershell Skriptes** sind zwei Dinge zu berücksichtigen. Zum einen verlangt Powershell oft eine **signierte Datei** und weigert sich ansonsten, diese auszuführen, zum anderen weigert sich Powershell, ein Skript nur durch Nennung des Namen auszuführen, auch wenn es im gleichen Verzeichnis liegt. So muss statt **build.ps1** **.\build.ps1** angegeben werden. Ein Verhalten, das zu Irritationen führen kann.



Bei der Ausführung mithilfe des Powershell Skriptes können dieselben Parameter verwendet werden, wie bei der Nutzung des im folgenden beschriebenen **GlobalTools**.

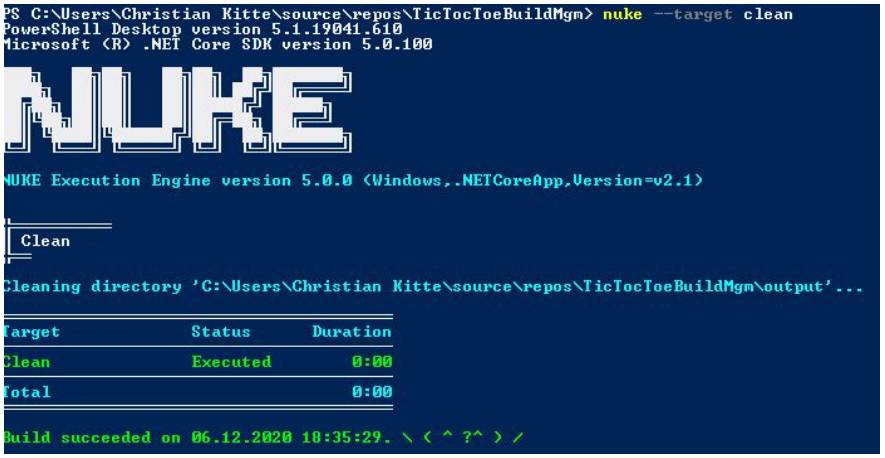
Ausführung mit dem GlobalTool

Es stellt sich als wesentlich einfacher heraus, den Build mit Hilfe des ganz zu Anfang installierten **GlobalTools** auszuführen.

Hierfür wird immer noch die **Powershell** genutzt, jedoch kommt nun kein Powershell Skript zum Einsatz, sondern das **GlobalTool**. Im einfachsten Fall reicht hierfür der Befehl **NUKE**, der den in der **Main Methode definierten Task** als Default ausführt:

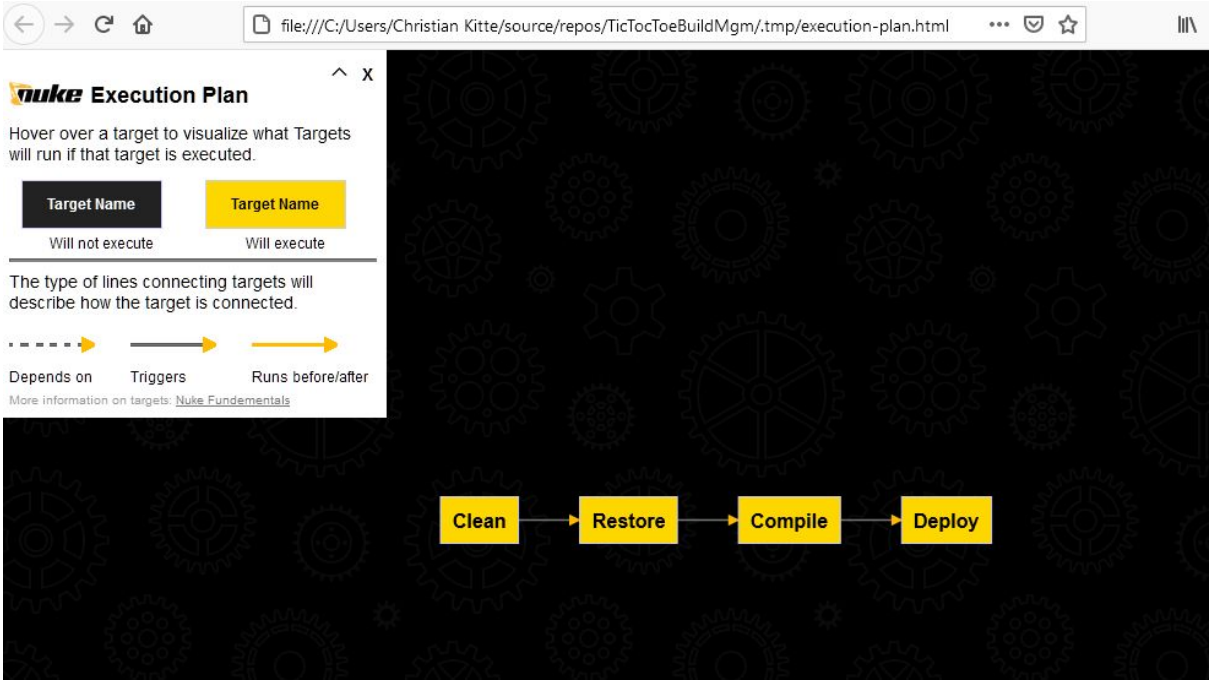
Für eine feinere Steuerung kann der Einsatz von Parametern nützlich sein. In der Regel ist die Syntax **NUKE [target] [arguments]**. Die wichtigsten sind:

--target X ⇒ hier wird explizit nur der oder die angegebenen Task' ausgeführt wie hier der Task **Clean**:



--configuration release bzw. **debug** ⇒ kompiliert den Sourcecode in der genannten Art als Release oder Debug.

Eine sehr hilfreiche Funktionalität ist die Fähigkeit von **NUKE**, einen **visuellen Ausführungsplan** auszugeben. Dies geschieht mit dem Parameter **--plan**:



Anmerkungen

Für mich persönlich ist **NUKE** das erste Buildmanagement, mit welchen ich mich - vor allem praktisch - beschäftigt habe.

Die hier vorliegende Arbeit ist nur ein erster Einstieg in die Funktionalität, welche **NUKE** zu bieten hat. Eine detaillierte Betrachtung aller Funktionen kann und soll hier nicht gemacht werden.

Sinnvoll im Rahmen dieser Arbeit, jedoch zeitlich nicht umsetzbar, wäre die Beschäftigung mit den erweiterten Klassen und Funktionalitäten zur **automatisierten Selektion** der zum **Deployen notwendigen Artefakte** und deren automatisiertes Kopieren in ein Ausgabeverzeichnis. Zwar arbeitet meine eigene Lösung, müsste aber immer wieder angepasst werden. Auch die Durchführung von **Tests** habe ich auf Grund der Zeit aktuell noch nicht gemacht, vor allem auch, da ich keine **Tests** definiert habe.

Mein eigenes Fazit ist, dass ich **NUKE** gerne für mich einsetzen würde und versuche, dies bei neuen Arbeiten umzusetzen. Die Arbeit mit **NUKE** macht Spass, bringt sehr viel Erleichterung und verspricht weniger Fehler (und da spreche ich aus Erfahrung).