Aufgabe

Bei der Aufgabe handelt es sich diesmal darum, sich mit einem beliebigen Continuous Integration System näher zu beschäftigen und es nachweisbar mit etwas Logik zum Laufen zu bringen.

Auswahl des Tools

Bei der Auswahl des Tools habe ich mich für GitHub Actions entschieden. Zu der Auswahl des Tools haben mich verschiedene Gründe bewegt:

- GitHub ist in der Praxis weit verbreitet im Einsatz
- GitHub Actions sind tief in GitHub integriert und unterstützen so dessen Infrastruktur
- GitHub Actions sind grundsätzlich sprachunabhängig
- GitHub Actions kann für nicht private Repositories kostenlos genutzt werden
- GitHub Actions kann so hervorragend auch für eigene Projekte genutzt werden

Verwendetes Beispielprojekt

Als Beispielprojekt zur Demonstration des Tools verwende ich das bereits aus den vorherigen Aufgaben bekannte Spiel **TicTacToe** in der um NUKE erweiterten Version. Um es von den vorherigen Abgaben zu separieren habe ich es in ein neues Repository auf **GIT**[1] ausgelagert.

Sämtliche hier beschriebenen Arbeiten, darunter auch das Script, können dort direkt im Code eingesehen werden. Im folgenden gehe ich auf das Programm selbst nicht näher ein, da es nicht im Fokus dieser Arbeit liegt.

Angestrebte Ergebnisse

Es soll für das Programm ein automatisierter Build erstellt werden, der bei jedem Push oder Pull Request im Master ausgeführt wird. Im Anschluss sollen die erstellten Artefakte zum Download verfügbar sein.

Hierbei möchte ich mindestens zwei Variationen haben. Zum einen ein Workflow, der nativ die .NET Core Anwendung kompiliert, zum anderen möchte ich die mit NUKE erstellte Konsolenanwendung nutzen. Auch hier sollen die Artefakte im Anschluss zur Verfügung gestellt werden.

Der Erfolg der Ausführung soll zudem durch einen Badge innerhalb der ReadMe erkennbar sein. Zudem können alle Ergebnisse natürlich direkt in GitHub eingesehen werden.

[1] https://github.com/ChristianKitte/TicTocToeCI

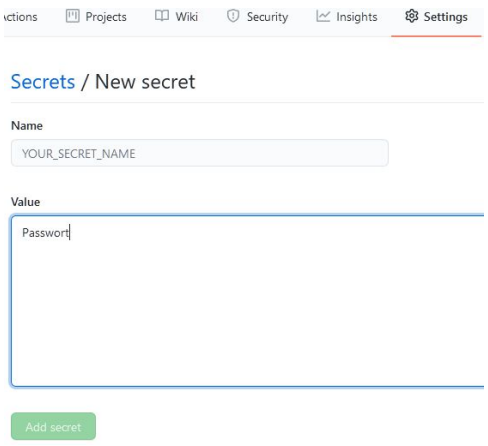
Systematik der GtiHub Actions

GitHub Actions stellen Workflows unter Kontrolle von GitHub dar, die auf GitHub und hier innerhalb der Repositories gehostet werden, auf die sie grundsätzlich abzielen.

GitHub Actions (im Folgenden werden diese zur Abgrenzung zu den weiter unten eingeführten Actions als Workflows bezeichnet) werden unabhängig von den in Repositories verwendeten Sprachen textuell mit einem Yamel Konfigurationsfile konfiguriert und weisen eine feste und dokumentierte[2] Ordnung auf. Jeder Workflow wird durch eine Yamel Datei definiert. Diese Dateien finden sich im Tab Code in dem von GitHub angelegten Ordner .github/workflows wieder.

Für jedes Repository können beliebig viele Workflows angelegt werden. Zur Orchestrierung der Workflows wird zum einen ein Event basiertes System eingesetzt, bei dem jeden Workflow Events zugewiesen werden, bei dem eine Ausführung startet. Darüber hinaus können Workflows andere Workflows ausführen und dessen Ergebnisse nutzen oder ihre eigenen weiter geben. Die kann auch über den Grenzen eines Repositories oder GitHub hinaus geschehen.

Notwendige Zugangsdaten können innerhalb eines Repository in einem speziellen, geschützten Bereich hinterlegt und Parameter basiert genutzt werden. Zugangsdaten erscheinen so niemals im Code und sollten auch niemals - auch nicht für Testzwecke - dort hart codiert werden.



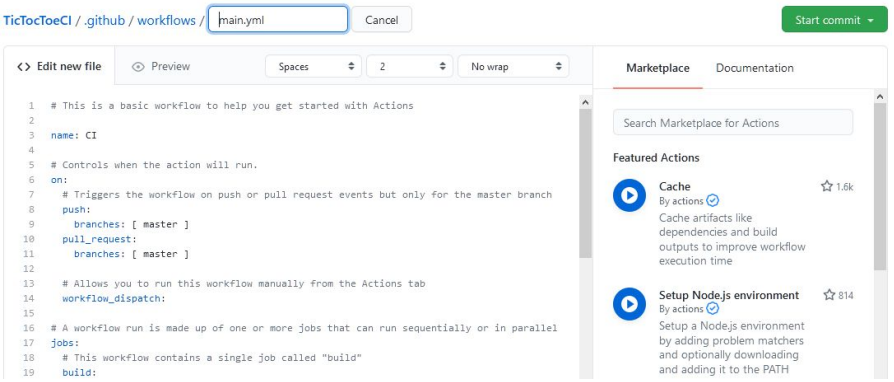
Wie oben zu sehen, erfolgt die Einstellung etwas versteckt über Settings - Secrets anstatt wie naheliegender über den Tab Security. In diesem Beispiel könnte man auf die zu schützenden Daten innerhalb der Yaml Datei mit ${{ YOUR_SECRET_NAME }}[3] zugreifen.

[2] https://docs.github.com/en/free-pro-team@latest/actions
[3] https://docs.github.com/en/free-pro-team@latest/actions/reference/encrypted-secrets

Die Nutzung der Workflows ist für öffentliche Repositories frei, private verfügen über einen begrenzten Workload und werden bei dessen Überschreitung nach Einsatz abgerechnet.

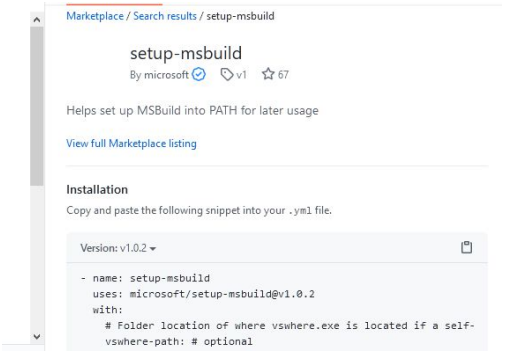
Ein zentraler Kern ist die Verfügbarkeit und Verwendung von Actions. Bei Actions im Sinne von GitHub handelt es sich um ein kleines Stück Software, welche eine bestimmte Funktionalität anbietet. Actions sind frei auf dem Marktplatz[4] verfügbar und können auch selbst erstellt und veröffentlicht werden.

Eine Action wird über einen einfachen, textlichen Eintrag in der Yamel Datei eingefügt. Es sind keinen Installationen erforderlich. GitHub unterstützt die Erstellung durch einen eigenen Editor mit teilweiser Codeunterstützung wie unten zu sehen.



Auf der rechten Seite werden verfügbare Actions angezeigt, auf der linken Seite eine allgemeine Vorlage. Wie ganz oben im Bild zu sehen ist, kann der Name der Datei selbst bestimmt werden und ermöglicht so eine individuelle Anpassung an die eigenen Bedürfnisse.

Durch einfaches anklicken einer Action werden die erforderlichen Einträge für die Nutzung, wie unten zu sehen, direkt angezeigt und können so in die Konfigurationsdatei übernommen werden:



[4] https://github.com/marketplace

Systematik der Workflows

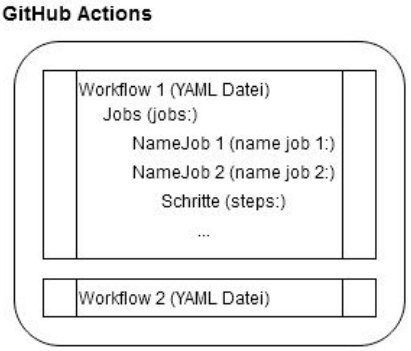
```
name: CI for Node.js Project
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
jobs:
  build:
    runs-on: ubuntu-latest
    name: Build and Test
    steps:
```

Für jeden Workflow existiert eine frei benennbare Yamel Datei, welche den Workflow darstellt und konfiguriert. Jeder Workflow verfügt über einen frei wählbaren Name (name:), welcher unabhängig vom Dateinamen ist.

Weiter werden für jeden Workflow das oder die Ereignisse, zu dem er starten (on:) soll festgelegt (push:, pull_request:, ...).

Ein Workflow verfügt über eine Auflistung von Jobs (jobs:). Für jeden Job wird eine virtuelle Maschine (s.u.) bestimmt, auf der er laufen soll (runs-on:). Jeder Job (build: Build and Test) ist frei benennbar und wird durch eine Auflistung von Schritten (steps:) definiert.

Somit ergibt sich für GitHub Actions / Workflows die folgende Übersicht:



Ausführung der angelegten Workflows

Besonders für den Einstieg ist die Ausführung der Workflows komplizierter zu erfassen, da nicht besonders hierauf eingegangen wird.

Grundsätzlich wird ein Workflow ausgeführt, indem der zu bearbeitende Code des Repositories zunächst ausgecheckt[5] wird und dann zu einer jeweils neu angelegten virtuellen Maschine herunter geladen wird (Download). Auf dieser Maschine wird dann der Workflow abgearbeitet.

Es stehen verschiedene Systeme zur Auswahl, wobei windows-latest bzw. ubuntu-latest die gebräuchlichsten sind. Dies bedeutet, dass Befehle sich an das System anpassen müssen (z.B. bei Pfadangaben).

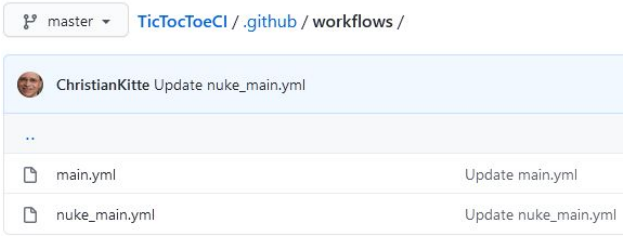
Vereinfacht kann man sich so die Yamel Dateien in Teilen als Skripte für den Aufruf auf dem Gastsystem vorstellen. Befehle wie mkdir, echo oder cp können unter Beachtung der Syntax und Besonderheiten direkt zur Ausführung gebracht werden.

[5] GitHub Sprachgebrauch

Gleichzeitig bedeutet dies, das Ergebnisse und Artefakte innerhalb dieser Maschinen erzeugt werden und nicht direkt im Repository. Um Ergebnisse zurück zu gewinnen, müssen diese zunächst aus der virtuellen Maschine wieder in das Repository hochgeladen werden (Upload).

Meine Umsetzung

Im folgenden möchte ich gerne meine zwei Lösungen vorstellen. Die zugehörigen Dateien finden sich im Workflow Ordner. Hierbei ist in main.yml der MSBuild, in nuke_main der NUKE Build definiert.



Workflow 1 mit MSBuild

Da die Datei innerhalb von GitHub frei verfügbar ist, gehe ich hier nur auf einige Besonderheiten ein. Der Workflow ist gemäß den zuvor gemachten Ausführungen so eingestellt, dass er für den master Branch bei eine Push und Pull Request Anforderung ausgeführt wird. Als virtuelle Maschine wird windows-latest gewählt.

```
# Steps represent a sequence of tasks that wi
steps:
  # Checks-out your repository under $GITHUB_
  - uses: actions/checkout@v2

  - name: Setup MSBuild
    uses: microsoft/setup-msbuild@v1

  - name: Setup NuGet
    uses: NuGet/setup-nuget@v1.0.5

  - name: Restore NuGet packages
    run: nuget restore TicTocToe.sln

  - name: Build the Solution
    run: msbuild TicTocToe.sln /p:configurati

  - name: Upload a Build Artifact by MSBuild
    uses: actions/upload-artifact@v2.2.1
    with:
      name: Repository BackUp by MSBuild
      path: ${{ github.workspace }}
```

Mit der Action checkout wird der Code aus dem Repository ausgecheckt. Dieser Schritt ist immer notwendig.

In den nächsten Schritten wird zunächst MSBuild sowie NuGet (Packagemanager .NET) verfügbar gemacht.

Mit Hilfe von nuget restore werden alle Abhängigkeiten aktualisiert und ggf. notwendige Pakete geladen.

“Build the Solution” startet den eigentlichen Buildprozess. Die wichtigste Option ist configuration=Release.

Mithilfe der Action upload-artifact können Artefakte hochgeladen werden, damit sie aus dem Repository verfügbar sind.

In diesem Fall wird zunächst der gesamte Workspace unter dem Titel “Repository BackUp by MSBuild” hochgeladen. Anschließend werden, wie unten zu sehen, einige Dateien in ein gemeinsames Verzeichnis kopiert und im Anschluss dies neue Verzeichnis hochgeladen. Dies könnte man auch einfacher z.B. mit der dotnet CLI umsetzen.

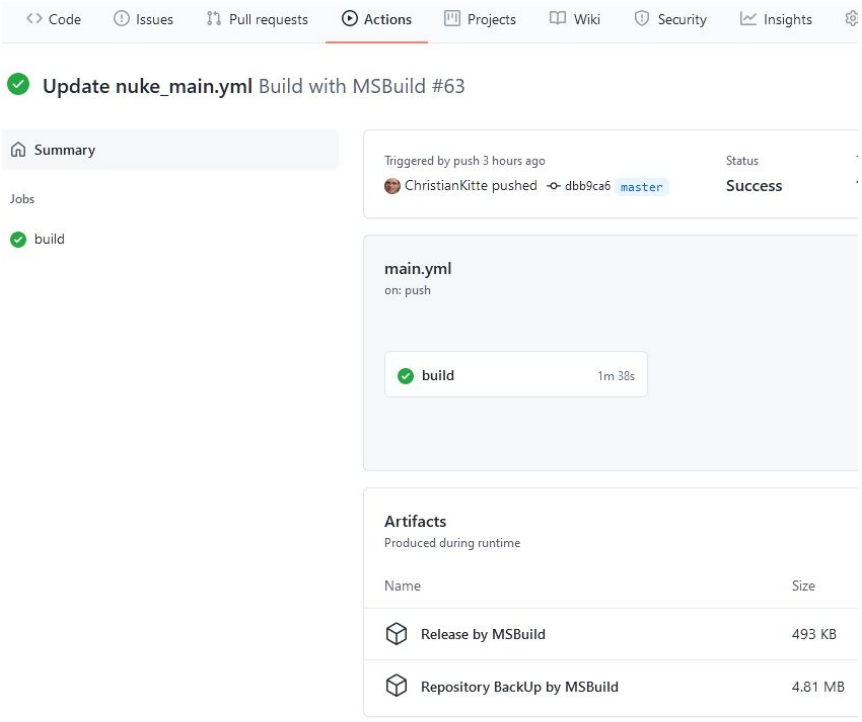
```
- name: collect artifacts
  run: |
    mkdir D:\a\TicTocToeCI\TicTocToeCI\Artifacts
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\Autofac.dll D:\a\TicTocToeCI\TicTocToeCI\Artifacts\Autofac.dll
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\TicTocLib.dll D:\a\TicTocToeCI\TicTocToeCI\Artifacts\TicTocLib.dll
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\TicTocToe.dll D:\a\TicTocToeCI\TicTocToeCI\Artifacts\TicTocToe.dll
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\TicTocToe.dll.config D:\a\TicTocToeCI\TicTocToeCI\Artifacts\TicTocToe.dll.config
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\TicTocToe.exe D:\a\TicTocToeCI\TicTocToeCI\Artifacts\TicTocToe.exe
    cp D:\a\TicTocToeCI\TicTocToeCI\TicTocToe\bin\Release\netcoreapp3.1\TicTocToe.runtimeconfig.json D:\a\TicTocToeCI\TicTocToeCI\Artifacts\TicTocToe.runtimeco
    cp D:\a\TicTocToeCI\TicTocToeCI\tutorial\Tutorial.html D:\a\TicTocToeCI\TicTocToeCI\Artifacts\Tutorial.html

- name: Upload a Build Artifact 2
  uses: actions/upload-artifact@v2.2.1
  with:
    name: Release by MSBuild
    path: D:\a\TicTocToeCI\TicTocToeCI\Artifacts
```

Wie zu sehen benutze ich hier die zuvor angesprochene Möglichkeit, Befehle direkt auf der virtuellen Maschine ausführen zu können, indem ich zunächst mit mkdir ein Verzeichnis erstelle, danach Dateien mit cp in dieses Verzeichnis kopiere und zuletzt dies Verzeichnis als ganzes hochlade. Hierbei sieht man, dass zur Bezeichnung des Objektes das Schlüsselwort path zu nutzen ist und name lediglich ein frei vergebbarer Name ist.

Um Zugriff auf die Dateien zu erhalten, kann über den Tab Actons der Workflow ausgewählt werden. Wie unten zu sehen erscheint ein Graph mit den Workflows (hier nur einer) sowie das Panel Artifacts, welches einen Download der hochgeladenen Dateien anbietet.

Interessant ist hierbei die Tatsache, das der Download auf den eigenen Rechner, das Entpacken und der Start selbst, den Norton Downloadschutz zu einer Warnung veranlasst.



Workflow 2 mit NUKE

Bei dem zweiten Workflow möchte ich die in dem Programm im Rahmen der zuvor gegebenen Aufgabe (Buildmanagement) gemachte Integration von NUKE nutzen.

Laut der Dokumentation von NUKE ist es möglich, u.a. bei GitHub Actions mit Hilfe einer Annotation der Buildklasse[6] die benötigte Yaml Datei automatisch auf Basis des Buildprojektes generieren zu lassen, was ein gewaltiger Vorteil ist. Das bei diesem Projekt jedoch aufgetretene Problem ist, dass die benötigten Dependencies scheinbar noch nicht verfügbar waren. Auch eine Umstellung von .NET Core 5 auf 3.1 brachte hier keine Lösung. Daher wurde auf diesen Weg verzichtet.

Um Wiederholungen zu vermeiden, gehe ich nur auf den unterschiedlichen Teil des Workflows ein.

```
- name: Setup .NET Core SDK
  uses: actions/setup-dotnet@v1.7.2
  with:
    dotnet-version: 3.1.x

# here comes the tricky part
- name: Build mit NUKE ausführen
  run:
    ./build.cmd --target Compile --configuration Release
```

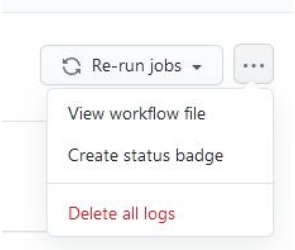
Die Action setup-dotnet ist ein Setuptool für das Dotnet Framework, was zuvor nicht notwendig war.

mit dem Befehl ./build.cmd wird direkt die Konsolenanwendung, welche aus der Verwendung von NUKE resultiert, gestartet.

Wichtig ist hier die Option --configuration Release um als Release zu kompilieren, sowie --target Compile, um explizit mit diesen Schritt zu starten. Dies schließt das Target Deploy aus, in welchem Dateien direkt kopiert werden und welches hier zu einem Fehler führen würde.

Setzen der Badges

Um den bekannten Badge zu nutzen, kann man sich den hierfür benötigten Code nach zumindest einen erfolgreichen Durchlauf des Workflows generieren lassen.



Hierfür geht man via dem Tab Action zu den verfügbaren Workflows, wählt einen Workflow aus und benutzt das oben rechts befindliche Menü.

Hierauf wird ein Dialog geöffnet, mit dessen Hilfe man sich den entsprechenden Markdown für den jeweiligen Workflow generieren lassen kann.

[6] https://nuke.build/docs/authoring-builds/ci-integration.html

Die Nutzung eines Badges hat den Vorteil, anderen aber auch sich selbst über die erfolgreiche Kompilierung zu unterrichten. In Verbindung mit der Option, die Kompilierung mit Hilfe verschiedener Events[7] zu starten oder der Durchführung von Tests, bietet sich hier eine mächtige Möglichkeit, einfach und sicher den aktuellen Zustand einer Software zu visualisieren.

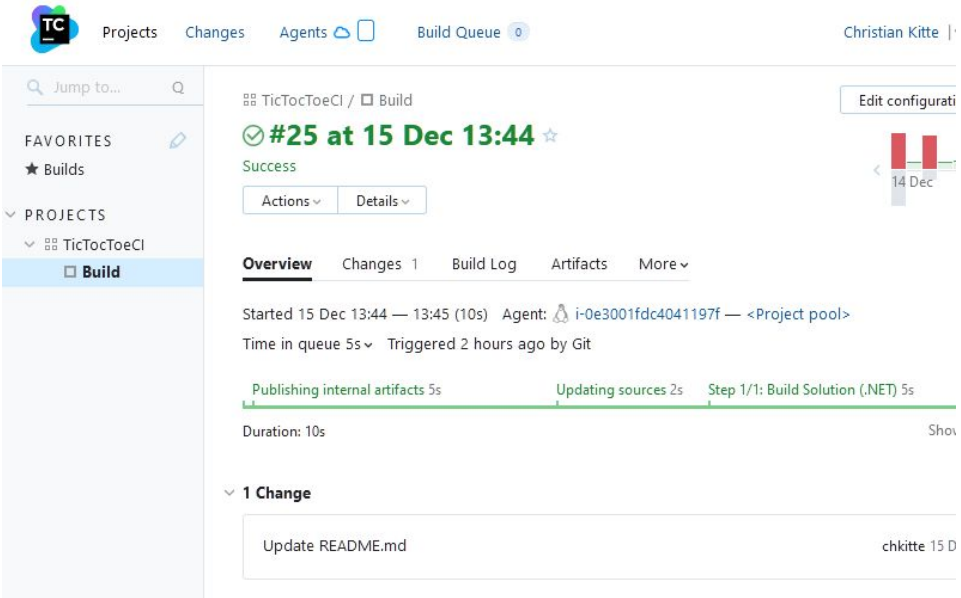
Anmerkungen

Für mich persönlich war die Arbeit mit GitHub Actions sehr lehrreich, aber auch eine Herausforderung. Zwar ist der Einstieg sehr leicht, jedoch wird vor allem in der Dokumentation einiges an Grundwissen vorausgesetzt. Hier bringt auch die Suche auf den üblichen Kanälen wie Stackoverflow nur bedingt weiter.

So habe ich beispielsweise besonders viel Zeit mit dem Problem verbracht, wie ich meine Daten direkt in ein Verzeichnis des Masters zurückkopieren kann. Letztlich habe ich es aus Zeitgründen nicht weiter verfolgt.

Trotz dessen könnte ich mir für die Zukunft vorstellen, GitHub und seine Möglichkeiten wesentlich mehr auch für CI zu nutzen. Auf jeden Fall glaube ich, mit dem bei GitHub Actions erworbenen Wissen, mich auch durchaus in andere Systeme einfinden zu können.

So habe ich beispielsweise Jetbrains TeamCity in der Cloudversion (Beta) recht schnell erfolgreich am laufen bekommen. Im Gegensatz zur lokalen Version vereinfacht die Cloudversion vieles u.a. auch im Umgang mit GutHub. Es ist visuell und im kleinen kostenlos. Ein Blick lohnt sich.



[7] https://docs.github.com/en/free-pro-team@latest/actions/reference/events-that-trigger-workflows