REFACTORING MIT NUGET UND JENKINS

Stark im Doppelpack

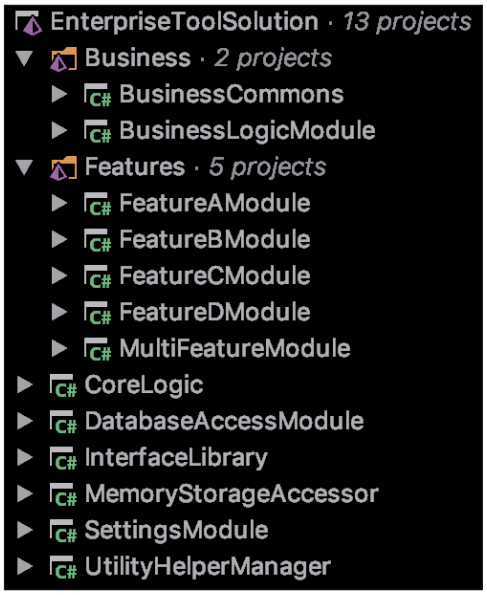
Im Zusammenspiel bieten NuGet und Jenkins ungeahnte Möglichkeiten zur Verbesserung der Softwarequalität.

Projekte beginnen üblicherweise klein und wachsen im Lauf der Jahre und Jahrzehnte. Auf Dokumentation und Softwarequalität wird dabei selten Wert gelegt. Dies hat zur Folge, dass nach einiger Zeit Wildwuchs entsteht. Einzelne Funktionen werden nach Bedarf (und Belieben) kopiert und verändert. Irgendwann sind die verantwortlichen Entwickler nicht mehr da, und die Klone geraten in Vergessenheit. Dann wird ein Fehler an einer Stelle gefunden, aber nur dort behoben, denn an die Kopien dieser Funktionalität denkt keiner mehr. Wer schon einmal versucht hat, in einem „historisch gewachsenen" Projekt Maßnahmen zur Verbesserung der Qualität einzuführen, kennt die Situation: Das Testen von Funktionalität ist schwierig, und statische Codeanalysen führen eher zu Tränen als zu Verbesserungen. Deswegen befasst sich dieser Artikel damit, wie gezieltes Refactoring und der Einsatz von NuGet und Jenkins derartige Situationen verbessern können.

NuGet ist derzeit der gebräuchlichste Paketmanager für .NET. Im Jahr 2010 ursprünglich als NuPack veröffentlicht, ist diese Open-Source-Paketverwaltung inzwischen in gängige .NET-IDEs fest integriert und sogar Bestandteil des Kommandozeilenwerkzeugs dotnet. Ich bin mir sicher, dass Sie in Ihrer .NET-Laufbahn bereits mit NuGet in Berührung gekommen sind, nämlich um Bibliotheken oder Bestandteile von .NET (Core) in ein Projekt zu integrieren. Aber mit NuGet ist weit mehr möglich. Sie können mit wenig Aufwand eigene Pakete erstellen und interne Netzlaufwerke als Quelle für NuGet-Pakete verwenden. Mit einer Jenkins-Instanz können Sie dies automatisieren und mit nur wenigen Schritten die Qualität Ihrer Software spürbar verbessern. Aber eines nach dem anderen.

Teilen und herrschen

Der Artikel nimmt immer wieder Bezug auf eine fingierte C#-Solution, die den Namen *ETS – Enterprise Tool Solution* trägt. Sie soll ein Projekt repräsentieren, wie es nach vielen Jahren der Entwicklung in einem Unternehmen entstanden sein könnte (siehe Bild 1). Den Quellcode dazu finden Sie auf meiner GitHub-Seite [1].



Projektübersicht der Enterprise Tool Solution (Bild 1)

In dieser Solution gibt es eine Funktion, die ein Byte-Array in eine Ganzzahl umwandelt – in fünf verschiedenen Ausprägungen, verteilt auf vier verschiedene Projekte. Dafür existieren keine Tests, keine Dokumentation, und Fehler sind auch noch mit dabei.

Es gibt mehrere Möglichkeiten, diese Situation zu verbessern. Man kann in dieser Solution ein Projekt erstellen, das die „allgemeingültige" Funktionalität beinhalten soll. Projekte dieser Art sind oft als „Common"-Projekte bekannt. In ein solches wird diese Funktion verschoben, und alle Projekte, die diese Funktion benötigen, referenzieren es. Dieses Vorgehen bringt allerdings einige Nachteile mit sich:

- Wenn es in einer Solution ein Common-Projekt gibt, tendiert man dazu, dieses als Mülltonne für all die Dinge zu verwenden, die sonst nirgendwo hineinpassen. Möglicherweise haben Sie schon entdeckt, dass es bereits ein Utility-HelperManager-Projekt und ein BusinessCommons-Projekt gibt. Beide Projekte lassen bereits vom Namen her erahnen, dass sie so ziemlich alles beinhalten, was sonst nirgendwo hingepasst hat.
- Irgendwann wird fachliche Logik in einem Common-Projekt landen, beispielsweise wenn aus Zeitgründen oder mangels besseren Wissens Code in das Common-Projekt verschoben wird, weil er an mehreren Stellen verwendet werden soll. Das ist sehr problematisch, da ein Projekt mit allgemeingültiger Funktionalität generell keine fachliche Logik enthalten sollte. Im schlimmsten Fall kann dies zu zirkulären Abhängigkeiten führen, die ein Kompilieren der Solution verhindern. Spätestens dann ist ein größeres Refactoring notwendig.
- Was ist mit anderen Projekten, die diese Funktionen verwenden wollen? In diesem Fall müsste man eine – lokal selbst gebaute – DLL in die anderen Projekte kopieren. Damit hätte man dann eine sehr ähnliche, aber noch schlimmere Situation geschaffen: Im Gegensatz zum kopierten Code in der Solution würden einzelne DLLs kopiert, bei denen keiner wüsste, was genau in ihnen steckt.

Netzlaufwerk als Kontrollmechanismus

Die Verwendung eines Netzlaufwerks für selbst erstellte NuGet-Pakete begünstigt einen weiteren Anwendungsfall: Neben den eigenen Paketen lassen sich auch Pakete von NuGet.org herunterladen und auf das Netzlaufwerk kopieren. So können Sie kontrollieren, welche Fremdbibliotheken in der eigenen Software verwendet werden, um einen möglichen Wildwuchs zu vermeiden. Darüber hinaus lassen sich firmeninterne Vorgaben, zum Beispiel welche Open-Source-Lizenzen erlaubt sind, auf diese Weise umsetzen. Dafür müssen Sie übrigens nicht jedes Paket einzeln mit *nuget add* hinzufügen. Sie können stattdessen das folgende Kommando ausführen:

```
nuget init C:\local\packages \\your\share\packages
```

Dann werden alle NuGet-Pakete, die sich im Verzeichnis *C:\local\packages* befinden, in Ihr Netzlaufwerk kopiert.

Also muss es anders gehen. Das Projekt wird in eine eigene Solution extrahiert. Zudem bekommt diese Solution einen eindeutigen Namen. Es soll eine Konvertierungsbibliothek werden, also heißt sie *Der Konverter*. Dieses Vorgehen beseitigt gleich zwei der oben angesprochenen Nachteile. Zum einen ist der Inhalt über den Namen klar definiert und es wird – hoffentlich – nichts anderes darin landen. Zum anderen ist der Code komplett von Fachlichkeit entkoppelt.

Bleibt das Problem der Verteilung der fertigen Bibliothek. Wird sie lokal von jemandem kompiliert und an das Team verteilt, so ist das Problem umgangen, aber nicht behoben. Also sollte sie zentral erstellt werden. Und am besten als NuGet-Paket, denn dann weiß das Team bereits, wie sie eingebunden werden kann.

Packen in fünf Schritten

Die nachfolgenden Schritte werden größtenteils über die Kommandozeile ausgeführt – im Verzeichnis der neuen Konverter-Solution. Für die Erstellung eines NuGet-Pakets ist eine Spezifikation notwendig, eine **.nuspec*-Datei. Diese wird mit dem Kommandozeilenbefehl von NuGet erstellt:

```
nuget spec Converter\Converter.csproj
```

Als Ergebnis kommt die Datei *Converter.csproj.nuspec* dabei heraus. Diese muss manuell angepasst werden, sodass sie aussieht wie in Listing 1 gezeigt.

All die Felder, die mit *$*-Ausdrücken gefüllt sind, füllt NuGet mit den entsprechenden Werten aus der *AssemblyInfo.cs* des Projekts. Es gibt noch etliche weitere Felder, die in der **.nuspec*-Datei stehen können, jedoch reicht für dieses Beispiel das Minimalset.

Als Nächstes wird das Projekt einmal kompiliert. Nutzen Sie dafür die Kommandozeile, da Sie diese auch für den danach folgenden Befehl wieder brauchen:

```
msbuild Converter\Converter.csproj /t:Rebuild
  /p:Configuration=Release
```

Wurde das Projekt erfolgreich kompiliert, kann man es jetzt in ein NuGet-Paket packen:

```
nuget pack Converter\Converter.csproj -Properties
  Configuration=Release
```

Fertig ist das erste eigene NuGet-Paket.

Der letzte Schritt ist das Deployment des Pakets. Und dies fördert erneut zwei Probleme zutage. Üblicherweise werden die Pakete auf NuGet.org hochgeladen, sodass sie über den herkömmlichen Weg heruntergeladen werden können. Handelt es sich bei dem extrahierten und nun gepackten Code jedoch um solchen, der nicht veröffentlicht werden darf, ist dies keine Option. Möglicherweise ist sogar der Zugriff auf NuGet.org aus Sicherheitsgründen gesperrt. Eine weitere Möglichkeit wäre dann der Einsatz eines eigenen (IIS)-Servers. Dieser kostet aber wiederum Geld für Hardware, Software und Wartung.

Glücklicherweise bietet NuGet die Möglichkeit, ein einfaches Netzlaufwerk als Paketserver verwenden zu können (vergleiche den Kasten Netzlaufwerk als Kontrollmechanismus). Ein Netzlaufwerk steht in den meisten Fällen zur Verfügung, sodass keine zusätzlichen Kosten entstehen. Das NuGet-Paket landet mit folgendem Befehl im Paketserver auf dem Netzlaufwerk:

```
nuget add Der.Konverter.1.0.0.nupkg -Source
  \\your\share\packages
```

Der Pfad ist das Ziel

Damit Sie das Paket nun verwenden können, teilen Sie NuGet mit, wo sich das Verzeichnis befindet. Auch dafür gibt es mehrere Möglichkeiten: Man kann die Informationen in den Konfigurationsdateien für den Computer, für den aktuel- ▶

Listing 1: Einfache Konfiguration eines NuGet-Pakets

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>$id$</id>
    <version>$version$</version>
    <title>$title$</title>
    <authors>$author$</authors>
    <owners>$author$</owners>
    <description>$description$</description>
    <releaseNotes>Unser erster Release.
      </releaseNotes>
    <copyright>Copyright 2019</copyright>
  </metadata>
</package>
```

len User oder für das aktuelle Projekt speichern. Da es wünschenswert ist, dass alle im Projekt die gleiche Konfiguration verwenden, ist es am besten, eine Konfigurationsdatei für das Projekt beziehungsweise die Solution zu erstellen und in das Versionskontrollsystem mit einzuchecken. Dazu legen Sie eine neue Datei *NuGet.Config* im Verzeichnis der Solution an und fügen folgenden Code hinzu:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
</configuration>
```

Der nächste Befehl fügt das Netzlaufwerk als neue Quelle hinzu:

```
nuget sources add -Name "Ihr NuGet-Server" -Source
  "\\your\share\packages" -ConfigFile .\NuGet.Config
```

Das Ergebnis ist in Listing 2 zu sehen.

Natürlich können die XML-Einträge auch manuell geschrieben werden. Unterläuft Ihnen beim Schreiben allerdings ein Fehler, wie zum Beispiel ein fehlendes schließendes Tag, wird NuGet die Konfigurationsdatei einfach ignorieren. Deswegen rät Microsoft zur Verwendung des Kommandozeilenwerkzeugs [2]. Details über weitere Konfigurationsmöglichkeiten können Sie in der NuGet-Dokumentation nachlesen [3]. Aktuelle IDEs wie Visual Studio oder Rider bieten ebenfalls Möglichkeiten, diese Einstellungen ganz ohne Kommandozeile vorzunehmen.

Jenkins und Git

Bevor es mit Jenkins weitergeht, noch ein Hinweis. Im Nachfolgenden wird davon ausgegangen, dass als Versionskontrollsystem Git verwendet wird und eine Instanz von Jenkins bereits existiert oder installiert werden kann, auf der alle notwendigen Werkzeuge (Git, MSBuild, .NET Framework, SDK) vorhanden sind.

Darüber hinaus ist im Git-Repository ein Hook aktiviert, der bei jedem Push, der eintrifft, den Jenkins-Server benachrichtigt [4]:

```
#!/bin/bash
curl http://<JENKINS-URL>/git/notifyCommit?url=
  <URL des Git-Repositorys>
```

Listing 2: NuGet.Config mit lokalem Server

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="Ihr NuGet-Server"
      value="\\your\share\packages" />
  </packageSources>
</configuration>
```

Für diesen Endpoint muss in Jenkins zusätzlich das Git-Plug-in installiert sein [5].

Pakete vom Fließband

Jenkins hat den Ruf, hässlich und für viele Aufgaben ungenügend zu sein. GitLab [6], Circle-CI [7] oder Bamboo [8] gelten als vielseitiger und moderner. Doch in vielen Enterprise-Umgebungen stehen diese Werkzeuge aus unterschiedlichen Gründen nicht zur Verfügung – nicht als On-Premises-Lösung und erst recht nicht Cloud-basiert.

Verschafft man sich einen Überblick über mögliche kostenfreie CI-Server [9], so bleiben nicht viele Optionen übrig. Jenkins ist eine davon, bietet die breiteste Unterstützung für andere Systeme und ist zudem meistens schon im Unternehmen oder der Abteilung vorhanden.

Um gleich mit dem Vorurteil des schrecklichen Aussehens aufzuräumen: Mit dem Plug-in Blue Ocean [10] bekommt die Software einen neuen Anstrich, der zudem die Bedienung wesentlich erleichtert. Tatsächlich kann damit das NuGet-Paket in wenigen einfachen Schritten vollkommen automatisch erzeugt werden.

Im ersten Schritt starten Sie Blue Ocean über den Eintrag in der Menüleiste der Jenkins-Startseite. Alternativ können Sie auch über den URL *https://ihre-jenkins-url/blue* direkt darauf zugreifen.

Wenn dies das erste Mal passiert, werden Sie aufgefordert, ein neues Pipeline-Projekt anzulegen [11]. Wählen Sie den Eintrag *Git* aus und tragen Sie den Pfad zum Repository ein – sollten Sie in Ihrem VCS einen speziellen User für Jenkins angelegt haben, so tragen Sie hier zusätzlich die benötigten Credentials ein. Anschließend bestätigen Sie die Eingaben über den Button *Create Pipeline*. Über den Button *Neue Pipeline* oben rechts können zu einem späteren Zeitpunkt weitere Projekte definiert werden.

Jenkins hat das Projekt nun erfolgreich angelegt und fordert als Nächstes dazu auf, eine Pipeline für das Projekt zu definieren. Dazu geben Sie zunächst an, auf welchem Agenten diese Pipeline laufen soll. Für dieses Beispiel wählen Sie die Option *node* und vergeben als Label die Bezeichnung *master.* Jenkins legt bei der Installation automatisch einen Knoten mit dem Namen *master* für sich selbst an. Für weiteren Konfigurationsmöglichkeiten sei auf die Jenkins-Dokumentation [12] verwiesen.

Jetzt haben Sie die Möglichkeit, die Pipeline über einen Klick auf das Pluszeichen um eine Stage (= Phase) zu erweitern. Zuerst fügen Sie den Git-Checkout als Stage hinzu, denn Jenkins braucht schließlich etwas, das gebaut werden kann. Der Stage geben Sie zum Beispiel den Namen *checkout* und fügen als Step das Modul *Git* hinzu. Dort konfigurieren Sie den Pfad zu dem Repository, das Jenkins auschecken soll. Das Feld für den Branch-Namen bleibt frei, damit alle Branches gebaut werden. Die nächste Stage bekommt den Namen *build* und zwei Steps vom Typ Windows Batch Script spendiert. In Ersteren schreiben Sie den Befehl, der NuGet die benötigten Pakete laden lässt:

```
nuget restore
```

In den zweiten Step schreiben Sie – leicht abgewandelt – den *msbuild*-Befehl, der weiter oben schon verwendet wurde, um das Projekt zu bauen:

```
msbuild DerKonverter.sln /t:Rebuild
  /p:Configuration=Release
```

Anstelle des einzelnen Projekts wird gleich die gesamte Solution gebaut. Den Grund dafür wird der Artikel später erläutern. Im Anschluss muss noch das NuGet-Paket erstellt werden, also definieren Sie eine weitere Stage mit einem Windows Batch Script, in das Sie folgenden bekannten Befehl schreiben:

```
nuget pack Converter\Converter.csproj -Properties
  Configuration=Release
```

Zuletzt fügen Sie eine Stage mit einem Windows Batch Script hinzu, die das NuGet-Paket auf das Netzlaufwerk schiebt. Da zukünftig verschiedene Versionen des Pakets gebaut werden sollen – und nicht immer nur 1.0.0 –, wird der Befehl ein wenig modifiziert:

```
nuget add *.nupkg \\your\share\packages
```

Fertig ist die Build-Pipeline für das NuGet-Paket.

Da war doch noch was …

Wenn Sie die Konfiguration dabei belassen, wird so für jeden Branch und jeden Commit ein neues NuGet-Paket mit der gleichen Versionsnummer, die in *AssemblyInfo.cs* hinterlegt ist, erzeugt. Die NuGet-Pakete werden aber nicht deployt, denn wenn NuGet versucht, ein Paket hinzuzufügen, das bereits existiert, so quittiert es dies mit einer Meldung und macht nichts weiter.

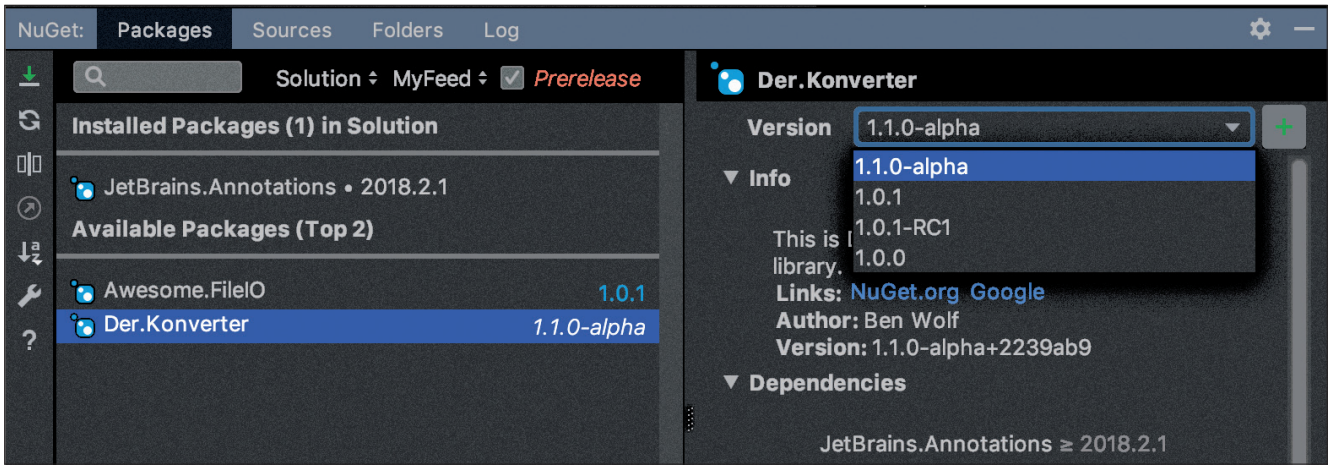
Hier ist von Vorteil, dass NuGet für die Pakete Semantic Versioning 2.0.0 [13] versteht. Somit können beispielsweise der Branch-Name sowie der Git-Hash oder die aktuelle Build-Nummer mit in den Namen gepackt werden. Ändern Sie den Befehl auf

```
nuget pack <project> -Suffix ${GIT_LOCAL_BRANCH}+${
  GIT_COMMIT}
```

ab, so erhalten Sie zum Beispiel *Der.Konverter.1.0.1-my-feature-branch+a7d4c42.nupkg*. Je nach Entwicklungsprozess, den Sie verwenden, können Sie darüber auch Release Candidates, Nightly Builds oder Betaversionen erstellen lassen. Diese Vorabversionen lassen sich ebenfalls über Ihre IDE installieren, sodass Sie Ihre neuen Funktionen testen können (siehe Bild 2) [14].

Apropos Testen

Einer der Gründe für das Extrahieren der Funktionalität in eine eigene Solution ist eine verbesserte Testbarkeit des Codes. Dafür legen Sie ein weiteres Projekt an, das den Namen *ConverterTest.csproj* bekommt. Das Konverter-Repository nutzt NUnit, daher verwenden auch die nachfolgenden Beispiele NUnit. Was Sie als Testframework einsetzen, ist nebensächlich. Wichtig ist einzig, dass Sie Ihre Software testen. Da auf Jenkins bereits die gesamte Solution gebaut wird, können Sie das Testprojekt (und zugehörige Tests) einfach einchecken und mit dem nächsten Build mitbauen lassen.



Übersicht über installierte und verfügbare NuGet-Pakete inklusive Prerelease-Versionen (Bild 2)

Damit ist es dann aber noch nicht getan, schließlich sollen die Tests auch ausgeführt werden. Hierfür legen Sie eine weitere Stage an, die das folgende Windows Batch Script beinhaltet:

```
[PathToNUnit]\nunit3-console.exe ConverterTest\bin\
  Release\ConverterTest.dll
```

Sie können Ihr Testwerkzeug entweder direkt auf den Build-Rechner kopieren oder Sie checken es mit in das Git-Repository ein. Das Ergebnis der Tests können Sie über eine zusätzliche Stage veröffentlichen, deren Step *Publish NUnit test result report* heißt. Wenn Sie bei Jenkins Plug-ins für ▶

Listing 3: Packen und veröffentlichen mit .NET Core

```
dotnet pack Converter\Converter.csproj
  /p:Configuration=Release
dotnet pack --version-suffix "your-branch+127a35e"
  Converter\Converter.csproj /p:Configuration=Release
dotnet nuget push Der.Konverter.1.0.0.nupkg -s
  \\your\share\packages
```

xUnit oder MSTest installieren, gibt es die Option zum Veröffentlichen der Testergebnisse ebenfalls für das jeweilige Test-Framework. Das Pattern für die Testdatei, die NUnit standardmäßig erzeugt, lautet *TestResult.xml*.

Ein Blick über den Tellerrand

Wenn Sie das oben beschriebene Szenario umgesetzt haben, haben Sie schon sehr viel erreicht:

- Sie haben eine kleinere und somit leichter wartbare Codebasis in Ihrer neuen Solution.
- Sie erzeugen automatisiert NuGet-Pakete für einzelne Branches und Versionen.
- Sie haben Tests für Ihren Code, die automatisch ausgeführt werden und bei Fehlern den Build fehlschlagen lassen.

Was Sie jetzt noch tun können:

- Mit Werkzeugen wie dotCover lässt sich die Testabdeckung des Codes überprüfen [15].
- Werkzeuge zur statischen Codeanalyse, wie zum Beispiel StyleCop mit Roslyn.Analyzers von Microsoft [16] oder ReSharper von JetBrains [17], können Ihnen einen Überblick über die Codequalität verschaffen und Ihnen dabei helfen, diese beständig zu verbessern oder eine gute Qualität zu bewahren.
- Wenn Sie Ihren Code gut kommentiert haben, können Sie für das API des NuGet-Pakets mit Werkzeugen wie Doxygen automatisch eine HTML-Dokumentation erstellen lassen [18].

Ein Wort zu .NET Core

Sollten Sie bereits .NET Core verwenden, so werden Sie beim Ausprobieren feststellen, dass ein Großteil der oben aufgeführten Befehle nicht funktioniert. Mit .NET Core haben sich einige Dinge geändert. Die Konfiguration Ihres NuGet-Pakets können Sie jetzt in den Einstellungen Ihres C#-Projekts vornehmen, die separate Konfigurationsdatei ist nicht mehr notwendig. Sie können Sie jedoch weiterhin verwenden, indem Sie den Pfad zur *.nuspec*-Datei im Projekt hinterlegen. Dann werden allerdings ausschließlich die Einstellungen dieser Datei verwendet.

Anstelle des NuGet-CLI verwenden Sie einfach dotnet. Die in diesem Artikel beschriebenen Schritte sind in ihrer .NET-Core-Ausprägung in Listing 3 zu sehen.

Der Befehl *dotnet pack* leistet mehr als *nuget pack*. Er stellt zuerst alle NuGet-Pakete wieder her (*nuget restore*), baut dann die Solution (*msbuild*) und erstellt letztlich das NuGet-Paket (*nuget pack*). Sie können diese Schritte aber auch innerhalb Ihres Projekts konfigurieren. Dann wird automatisch ein NuGet-Paket erzeugt, wenn das Projekt gebaut wird.

Zum Schluss möchte ich mich bei meinen ehemaligen Kollegen Johannes Broeker und Oliver Klein bedanken. Wir hatten das oben beschriebene Szenario in einem großen Projekt zusammen erfolgreich umgesetzt. Dies hat mich letztendlich zu diesem Artikel und auch zu meinem Vortrag „Golden Nu(g)Get“ [19] inspiriert. ■

[1] ETS – Enterprise Tool Solution, www.dotnetpro.de/SL1907NugetJenkins1
[2] Configuring NuGet behavior, www.dotnetpro.de/SL1907NugetJenkins2
[3] NuGet.Config Reference, www.dotnetpro.de/SL1907NugetJenkins3
[4] Git einrichten – Serverseitige Hooks, www.dotnetpro.de/SL1907NugetJenkins4
[5] Git-Plug-in in Jenkins, www.dotnetpro.de/SL1907NugetJenkins5
[6] GitLab, https://about.gitlab.com
[7] Circle-CI, https://circleci.com
[8] Atlassian Bamboo, www.dotnetpro.de/SL1907NugetJenkins6
[9] Comparison of continuous integration software, www.dotnetpro.de/SL1907NugetJenkins7
[10] Jenkins Blue Ocean, www.dotnetpro.de/SL1907NugetJenkins8
[11] Jenkins Pipeline, www.dotnetpro.de/SL1907NugetJenkins9
[12] Jenkins-Dokumentation, www.dotnetpro.de/SL1907NugetJenkins10
[13] Semantic Versioning, https://semver.org
[14] Erstellen von Vorabversionen von Paketen, www.dotnetpro.de/SL1907NugetJenkins11
[15] JetBrains dotCover, www.dotnetpro.de/SL1907NugetJenkins12
[16] StyleCopAnalyzers, www.dotnetpro.de/SL1907NugetJenkins13
[17] JetBrains ReSharper, www.dotnetpro.de/SL1907NugetJenkins14
[18] Doxygen, www.doxygen.org
[19] Benjamin Wolf, Golden Nu(g)Get – Eigene .NET-Pakete leicht gemacht mit NuGet und Jenkins, www.dotnetpro.de/SL1907NugetJenkins15

Benjamin Wolf
ist Architekt und Entwickler bei INNOQ. Er erträgt unsauberen Code nur schwer und scheut nicht vor umfangreichen Refactorings zurück. Als Coach und Trainer gibt er seine Vorstellung von Softwarequalität weiter.
benjamin.wolf@innoq.com

dnpCode A1907NugetJenkins