Automatisiertes T-SQL Unit Testing mit Jenkins, Docker tSQLt und dem JUnit Plugin

Kathegorie
SQL Server, Docker

Aaron
Priesterroth

In den vorausgehenden Artikel zu Jenkins, Docker und SQL Server haben wir uns bisher ausschließlich drauf beschränkt. In dem heutigen Artikel werden wir unsere multi-branch Pipeline mit zwei simplen tSQLt unit tests erweitern. Der einzige Unterschied zur Vorgehensweise des vorherigen Artikels ist, dass wir dieses Mal zusätzlich das JUnit Plugin für Jenkins benötigen.

Auch hier werden wir wieder das SQL Server 2017 Linux Image verwenden. Dabei kommt es leider zum Konflikt, da Linux nur mit Baugruppen kompatibel ist, die als SICHER (engl. SAFE) gekennzeichnet sind. tSQLt ist das nicht. Um die beiden Bausteine also trotzdem miteinander verwenden zu können, muss die tSQLtCRL als sicher erzeugt werden. Die Zugehörige Jenkins- und SSDT-Dateien können auf GitHub hier gefunden werden.

Vorbereitung der SQL Server Instanz und Datenbank

Damit wir tSQLt auf einer Instanz verwenden können, müssen wir zunächst die Instanz so konfigurieren, dass sog. **CLR assemblies** ausgeführt werden dürfen. Diese Einstellung kann mithilfe des folgenden Skripts durchgeführt werden:

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
GO
EXEC sp_configure 'clr enabled', 1
EXEC sp_configure 'clr strict security', 0
RECONFIGURE
GO
```


Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Als Nächstes können wir jetzt tSQLt herunterladen. Die Datei muss anschließend entpackt werden. Unter den erhaltenen Dateien befindet sich eine Datei mit dem Namen **tSQLt.class.sql**. Dieses Skript muss Du gegen die Datenbank, auf der Du die unit tests machen möchtest, ausführen. Zu Letzt muss **tSQLtCRL assembly** geöffnet werden, um zu verifizieren, dass PERMISSION_SET = SAFE gesetzt ist.

Die Jenkins Build Pipeline

Die in diesem Artikel verwendete Pipeline hat folgende Gestalt:

```
def BranchToPort(String branchName) {
def BranchPortMap = [
[branch: 'master' , port: 15565]
,[branch: 'Release' , port: 15566]
,[branch: 'Feature' , port: 15567]
,[branch: 'Prototype', port: 15568]
,[branch: 'HotFix' , port: 15569]
]
BranchPortMap.find { it['branch'] == branchName }['port']
}

def PowerShell(psCmd) {
bat "powershell.exe -NonInteractive -ExecutionPolicy Bypass -Command
\"\$ErrorActionPreference='Stop';$psCmd;EXIT \$global:LastExitCode\""
}

def StartContainer() {
PowerShell "If (\$((docker ps -a --filter \"name=SQLLinux${env.BRANCH_NAME}\").
Length) -eq 2) { docker rm -f SQLLinux${env.BRANCH_NAME} }"
docker.image('microsoft/mssql-server-linux').run("-e ACCEPT_EULA=Y -e SA_
PASSWORD=P@ssword1 --name SQLLinux${env.BRANCH_NAME} -d -i -p ${BranchToPort(env.
BRANCH_NAME)}:1433")
PowerShell "While (\$((docker logs SQLLinux${env.BRANCH_NAME} | select-string
ready | select-string client).Length) -eq 0) { Start-Sleep -s 1 }"
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1 -Q
\"EXEC sp_configure 'show advanced option', '1';RECONFIGURE\""
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1 -Q
\"EXEC sp_configure 'clr enabled', 1;RECONFIGURE\""
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1 -Q
\"EXEC sp_configure 'clr strict security', 0;RECONFIGURE\""
}

def DeployDacpac() {
def SqlPackage = "C:\\Program Files\\Microsoft SQL Server\\140\\DAC\\bin\\
sqlpackage.exe"
def SourceFile = "SelfBuildPipelineDV_tSQLt\\bin\\Release\\SelfBuildPipelineDV_
tSQLt.dacpac"
def ConnString = "server=localhost,${BranchToPort(env.BRANCH_
NAME)};database=SsdtDevOpsDemo;user id=sa;password=P@ssword1"
unstash 'theDacpac'
bat "\"${SqlPackage}\" /Action:Publish /SourceFile:\"${SourceFile}\" /
TargetConnectionString:\"${ConnString}\" /p:ExcludeObjectType=Logins"
}

node('master') {
stage('git checkout') {
timeout(time: 5, unit: 'SECONDS') {
checkout scm
}
}

stage('build dacpac') {
bat "\"${tool name: 'Default', type: 'msbuild'}\" /p:Configuration=Release"
stash includes: 'SelfBuildPipelineDV_tSQLt\\bin\\Release\\SelfBuildPipelineDV_
tSQLt.dacpac', name: 'theDacpac'
}

stage('start container') {
timeout(time: 20, unit: 'SECONDS') {
```

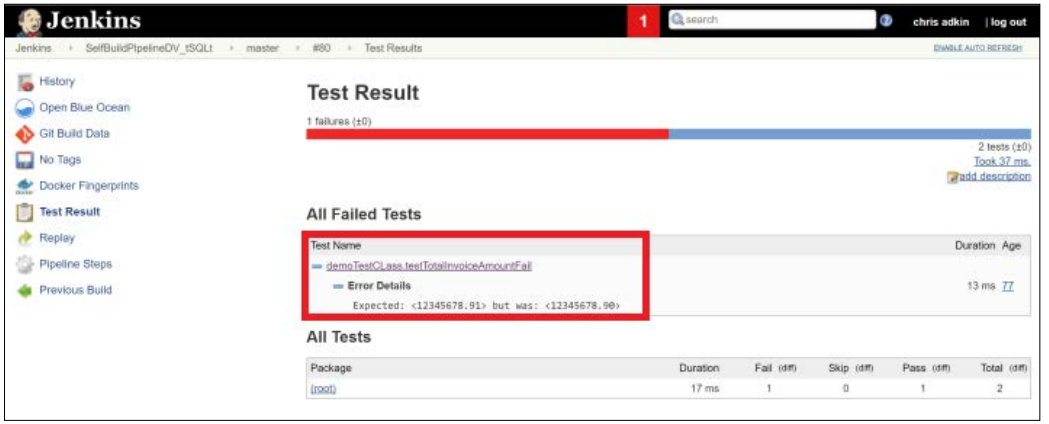

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

```
StartContainer()
}
}

stage('deploy dacpac') {
try {
timeout(time: 60, unit: 'SECONDS') {
DeployDacpac()
}
}
catch (error) {
throw error
}
}

stage('run tests') {
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1 -d
SsdtDevOpsDemo -Q \"EXEC tSQLt.RunAll\""
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1
-d SsdtDevOpsDemo -y0 -Q \"SET NOCOUNT ON;EXEC tSQLt.XmlResultFormatter\" -o
\"${WORKSPACE}\\SelfBuildPipelineDV_tSQLt.xml\""
junit 'SelfBuildPipelineDV_tSQLt.xml'
}
}
```

In diesem Beispiel für den Test mit tSQLt bildet die Klasse demoTestClass unsere Basis. Das zugehörige Schema beinhaltet zwei gespeicherte Prozeduren. Um beide Fälle abzudecken, werden wir zwei unit test betrachten, wobei einer der Tests erfolgreich absolviert wird und der andere fehlschlägt:

```
CREATE PROCEDURE [demoTestCLass].[testTotalInvoiceAmountPass]
AS
BEGIN
DECLARE @ActualTotal NUMERIC(10,2)
,@ExpectedTotal NUMERIC(10,2) = 12345678.90
SELECT @ActualTotal = ISNULL(SUM(Total), 12345678.90)
FROM [dbo].[Invoice];

EXEC tSQLt.AssertEquals @ExpectedTotal , @ActualTotal;
END;
GO

CREATE PROCEDURE [demoTestCLass].[testTotalInvoiceAmountFail]
AS
BEGIN
DECLARE @ActualTotal NUMERIC(10,2)
,@ExpectedTotal NUMERIC(10,2) = 12345678.91;

SELECT @ActualTotal = ISNULL(SUM(Total), 12345678.90)
FROM [dbo].[Invoice];

EXEC tSQLt.AssertEquals @ExpectedTotal , @ActualTotal;
END;
GO
```

Wenn wir diese Build Pipeline erstellen, erhalten wir wie erwartet einen Fehler. Verfolgen wir ihn, erkennen wir, dass es sich um den zuvor von uns konstruierten Fehler handelt. Also alles wie erwartet!



Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612

Aber wie funktioniert das eigentlich, dass wir unsere Ergebnisse so fein zurück verfolgen können? Obwohl es sich bei tSQLt und JUnit jeweils um Frameworks für unit test handelt, müssen wir Rücksicht auf den Aufbau von T-SQL und Java nehmen, um an die Ergebnisse unserer Tests zu gelangen. Um es einfach zu halten, liegt der entscheidende Punkt in der XML Ausgabe, die durch die Zeile erzeugt wird:

```
bat "sqlcmd -S localhost,${BranchToPort(env.BRANCH_NAME)} -U sa -P P@ssword1
-d SsdtDevOpsDemo -y0 -Q \"SET NOCOUNT ON;EXEC tSQLt.XmlResultFormatter\" -o
\"${WORKSPACE}\\SelfBuildPipelineDV_tSQLt.xml\""
```

Diese XML-Datei ist in einem Format, das identisch zu dem ist, das von JUnit verwendet wird, und wir haben damit also den Übergang zwischen den unterschiedlichen Frameworks geschaffen!

Mainzer Datenfabrik GmbH
Rotterdamer Platz 2, 55120 Mainz
info@madafa.de
madafa.de
+49 6131 3331612