ENTWICKLUNG VERTEILTER APPLIKATIONEN MIT TYE

Terraforming mit Muskelkraft

Verteilte Applikationen zu entwickeln kann schwierig sein. Project Tye von Microsoft ist angetreten, Entwicklern in dieser Hinsicht das Leben zu erleichtern.

Wie man eine Webapplikation containerisiert und eine verteilte Anwendung mittels Docker Compose lokal ausführt, wurde bereits in [1] vorgestellt. Von da aus ging es dann in [2] weiter in den Kubernetes-Cluster.

Das dort beschriebene Vorgehen ist nicht falsch: Mit der Containerisierung und der lokalen Ausführung zu beginnen und sich dann in einen Orchestrator wie Kubernetes zu steigern ist eine sinnvolle Evolution. Dass man als Zwischenschritt zum Beispiel Docker Compose verwendet, erleichtert das Starten und Stoppen der Applikation. Im Detail lässt das Verfahren aber doch einige Mängel erkennen. Zuerst ist das Schreiben von Definitionen für Docker Compose mühsam, vor allem, weil sich diese Definitionen nicht ohne Weiteres in Kubernetes-Manifeste überführen lassen. Und dann ist für jede Applikation, die man starten möchte, zuerst auch ein Docker-Image zu erzeugen. Von der fluffigen F5-Debugging-Experience ist man da schon ein gutes Stück entfernt.

Hier kommt Tye [3] ins Spiel, das sich als Tool vorstellt, das lokales Entwickeln und Testen von Microservices und verteilten Applikationen vereinfachen soll, mit der Option einer Bereitstellung in Kubernetes mit minimalem Aufwand.

Tye, zwei, eins, los!

Das hört sich gut genug an, um mal einen Blick zu riskieren. Man installiert Tye als globales .NET-Tool mittels

```
dotnet tool install -g Microsoft.Tye
  --version "0.11.0-alpha.22111.1"
```

Alternativ kann man mithilfe des *update*-Verbs auch eine bereits bestehende Installation aktualisieren. Vom Erfolg der Installation überzeugt man sich mit

```
tye --version
```

Es gibt auch eine Visual Studio Code Extension für Tye, was immer ein gutes Zeichen ist – diese ist unter dem Identifier *ms-azuretools.vscode-tye* bereitgestellt und lässt sich in der Suche über das Stichwort *tye* finden.

Derart ausgerüstet kann man sich direkt über das Quickstart-Tutorial hermachen [4]. Hier erstellt man eine MVC-Webanwendung als Frontend mit einem Web API als Backend und baut die klassische Wettervorhersage ein. Dem Tutorial kann man leicht folgen und es gelingt auch auf Anhieb. Der Kernaspekt, den man daraus mitnehmen kann, ist, dass Tye direkt mit einer .NET-Solution arbeiten kann. Will man mehrere Projekte simultan ausführen, führt man den Befehl

```
tye run
```

im Ordner mit der Solution-Datei aus. Dadurch werden alle Projekte gestartet und obendrein gibt es noch ein praktisches kleines Dashboard, das standardmäßig auf *localhost:8000* zu finden ist (vergleiche Bild 1).

Hier werden einige nützliche Informationen bereitgestellt. Auf den ersten Blick sieht man, welche Dienste gestartet wurden und ob diese als .NET-Projekt oder als Container gestartet wurden (Spalte *Type*). Durch einen Klick auf den Namen des Dienstes gelangt man zu einer Seite, die Metriken über den Dienst anzeigt. In der Spalte *Bindings* wird angezeigt, unter welchem URL der entsprechende Dienst gehostet wird. Anhand der Spalte *Restarts* kann man schnell erkennen, ob etwas nicht in Ordnung ist – ein Wert von 0 ist erstrebenswert. Wenn man genauer wissen möchte, was nicht läuft, klickt man auf den Link *Logs* und gelangt zu einer sich fortlaufend aktualisierenden Ansicht der Log-Ausgaben des Dienstes.

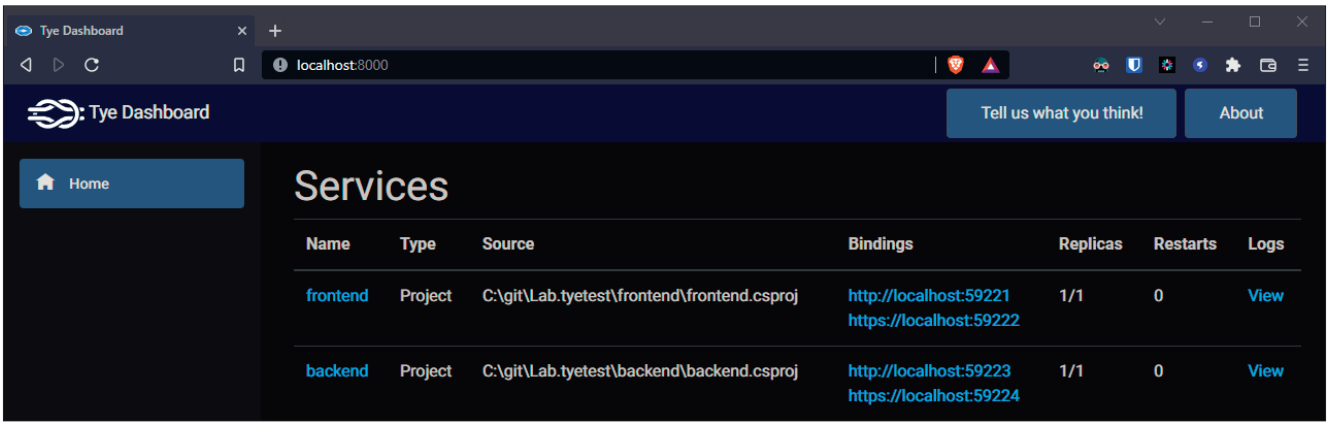
Der zweite Sweet Spot ist die Kommunikation zwischen Frontend und Backend. Nachdem man das NuGet-Paket *Microsoft.Tye.Extensions.Configuration* für die Konfiguration eingebunden hat, findet man den URL zum Backend einfach über den Namen des Dienstes mittels

```
builder.Configuration.GetServiceUri("backend");
```

Diesen kann man dann einfach für einen *HttpClient* als *BaseAddress* verwenden. Diese Service Discovery löst Tye über die automatische Injektion von Umgebungsvariablen in die laufenden Dienste.

Und ab in den Cluster damit!

Vom positiven Ergebnis der ersten Schritte derart ermutigt, wagen wir uns gleich an das zweite Versprechen von Tye: einfaches Deployment nach Kubernetes. Dafür braucht man natürlich erst mal einen Kubernetes-Cluster. Dafür gibt es



Das Tye-Dashboard (Bild 1)

verschiedene Möglichkeiten, etwa einen lokalen Cluster mittels Docker Desktop oder Minikube. Sehr einfach gelingt das Aufsetzen eines Clusters in Azure. Mit dem *az*-CLI ist ein Kubernetes-Cluster in Azure innerhalb weniger Minuten erstellt. Damit man später leicht aufräumen kann, packt man alles, was man braucht, in eine Ressourcengruppe:

```
az group create --location westeurope
  --resource-group tyetest
```

Damit man die Docker-Images an einem Ort ablegen kann, den der AKS-Cluster auch erreichen kann, erzeugt man zunächst eine Container-Registry

```
az acr create --resource-group tyetest
  --name tyetestcontainers --sku Basic
```

Damit Tye später Images pushen kann, die lokal gebaut wurden, muss man sich noch bei der Container-Registry anmelden:

```
az acr login -n tyetestcontainers
```

Die Erstellung des Clusters erfolgt mittels

```
az aks create --resource-group tyetest --name tyecluster
--attach-acr tyetestcontainers
```

Der Parameter *--attach-acr* sorgt dafür, dass die Identität des AKS-Clusters die *acrpull*-Rolle auf der Container-Registry erhält, um Images abrufen zu können. Nun heißt es warten, bis die Kommandozeile sich beruhigt hat. Um mittels *kubectl* mit dem Cluster kommunizieren zu können, sagt man dann noch

```
az aks get-credentials --resource-group tyetest
  --name tyecluster
```

Sehr komfortabel sind in diesem Zusammenhang die Kubernetes Tools für Visual Studio Code [5] – da kann man auch noch einmal nachlesen, wie die übrigen Tools zu installieren sind, die man für die Arbeit mit Kubernetes braucht. Hat man alles zusammen, kann man die Applikation einfach mittels

```
tye deploy -interactive
```

deployen. Zunächst gibt man dabei den Namen der zu verwendenden Container-Registry ein. In unserem Beispiel ist dies *tyetestcontainers.azurecr.io*. Anschließend beginnt Tye damit, die Docker-Images für die Applikation zu bauen. Dafür verwendet es eine rote Konsolenschrift, was beim ersten Anblick zu einem kleinen Schreck führen könnte. Wenn alles gut läuft, quittiert das CLI dies mit *"Deployed application 'lab.tyetest'"*

Wenn man das erste Staunen darüber abgeschüttelt hat, wie schnell das ging, kann man einmal vorsichtig durch das Schlüsselloch im Cluster spähen:

```
kubectl get pods
```

Dies zeigt, dass tatsächlich zwei Pods ausgeführt werden. Die Anwendung ist noch nicht öffentlich erreichbar, weil noch kein Ingress konfiguriert ist, aber man kann die Anwendung über ein Port-Forwarding direkt im Cluster testen:

```
kubectl port-forward svc/frontend 5000:80
```

Dann kann man im Browser *localhost:5000* aufrufen und sieht die Frontend-Applikation (vergleiche Bild 2).

Anhand der Meldung in der Kommandozeile erkennt man, dass man auch tatsächlich mit dem Cluster kommuniziert. Wenn man das Port-Forwarding beendet, indem man die Konsole schließt, meldet der Browser erwartungsgemäß, dass die Seite nicht erreichbar ist.

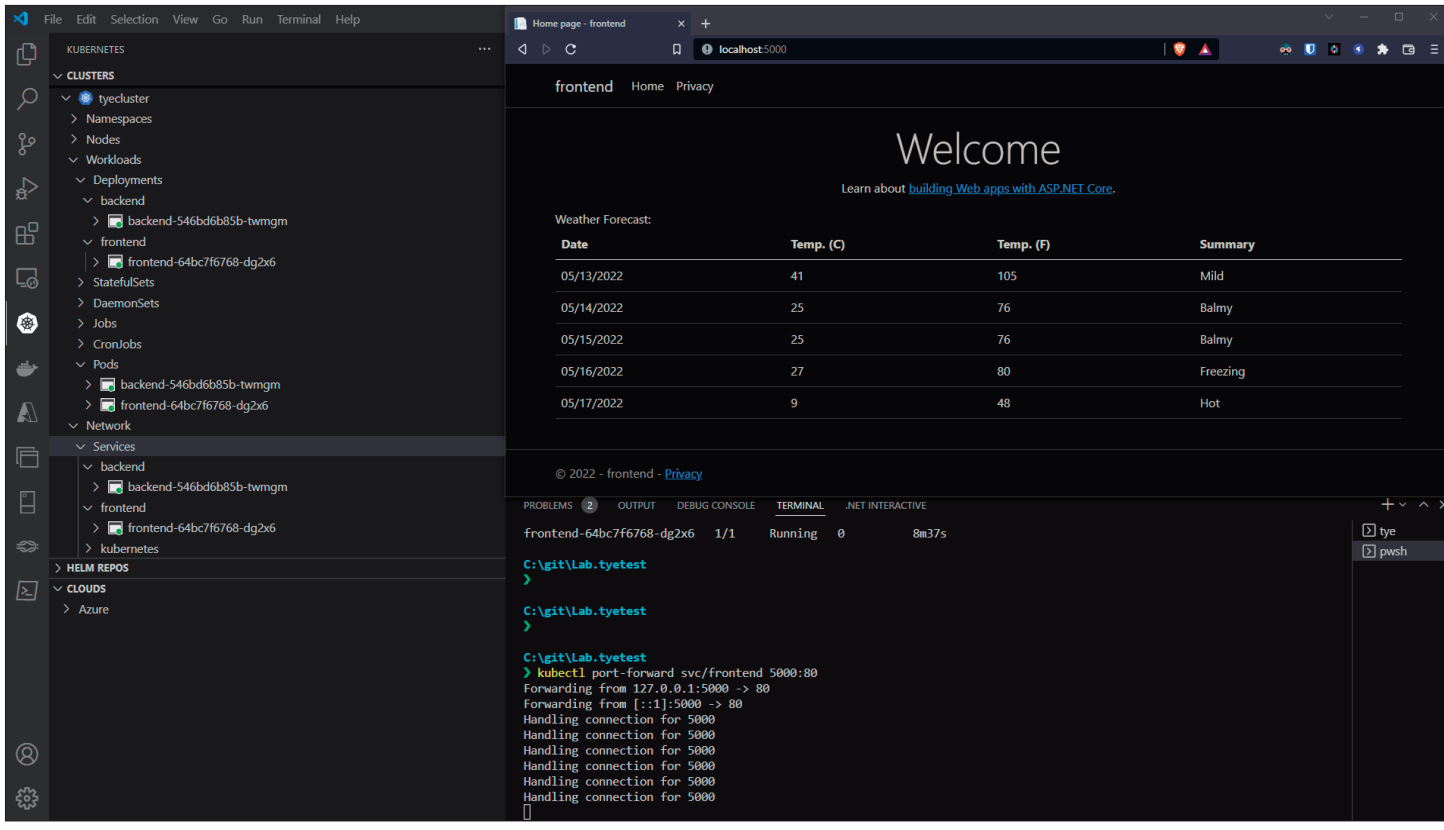
Um den Cluster wieder aufzuräumen, verwendet man den *undeploy*-Befehl

```
tye undeploy
```

und schon ist alles wieder dahin.

Noch mal von vorn und diesmal richtig(er)

Die bisherige Performance von Tye war schon recht beeindruckend. Wenn man etwas mehr Kontrolle über den Prozess ausüben möchte, kann man neben der Solution-Datei noch ▶



Die Testanwendung in AKS deployt (Bild 2)

eine Konfigurationsdatei für Tye anlegen. Diese lässt sich mit dem Befehl

```
tye init
```

erzeugen. Standardmäßig enthält die Datei im Abschnitt *services* lediglich Angaben über die Projekte, die eingebunden werden sollen. Über den Schlüssel *registry* kann man die Container-Registry spezifizieren, die verwendet werden soll – damit wird man schon mal eine lästige Eingabe beim Deployment los (vergleiche Listing 1).

Die Angaben zum *ingress* ab Zeile 10 sind manuell hinzugefügt worden. Diese helfen dabei, dass die Applikation auch ohne Port-Forwarding zu erreichen ist. Dazu wird während

Listing 1: Eine Konfigurationsdatei von Tye (tye.yaml)

```
1:  name: lab.tyetest
2:  registry: tyetestcontainers.azurecr.io
3:
4:  services:
5:  - name: frontend
6:    project: frontend/frontend.csproj
7:  - name: backend
8:    project: backend/backend.csproj
9:
10: ingress:
11:   - name: frontend
12:     rules:
13:       - path: /
14:         service: frontend
15:
16:       - host: tyetest.westeurope.cloudapp
            .azure.com
17:         path: /
18:         service: frontend
19:
20:
21:
22:   - name: backend
23:     rules:
24:       - path: /api
25:         service: backend
26:
27:       - host: tyetest.westeurope.cloudapp.azure.
com
28:         path: /api
29:         service: backend
```

des Deployments von Tye ein sogenannter Ingress Controller in den Cluster installiert. Dabei handelt es sich um einen NGINX-Webserver, der über spezielle Kubernetes-Konfigurationsdateien konfiguriert wird, die wiederum von Tye anhand der *ingress*-Konfiguration in der *tye.yaml* generiert werden. Gleichzeitig wird im Azure Load Balancer, der zu jedem AKS-Cluster gehört, eine öffentliche IP-Adresse erstellt, die mit dem NGINX-Ingress-Controller verbunden wird. Ruft man also diese IP auf, so landet man im Ingress Controller im Cluster, der den Traffic dann anhand von Pfaden oder Hostnamen an die Services im Cluster verteilt.

Für den Frontend-Service wird eine Ingress-Konfiguration mit dem Namen *frontend* angelegt, dieser Name taucht später im Cluster wieder auf. Die Regel in Zeile 13 bis 14 besagt, dass unter dem Root-Pfad der Service *frontend* angesprochen werden soll, und zwar unabhängig vom Host-Namen. Das führt dazu, dass man das Frontend später unter Angabe der öffentlichen IP-Adresse des Ingress Controllers erreichen kann. Für das Backend wird in Zeile 22 bis 25 dasselbe konfiguriert, allerdings hier mit dem Pfadzusatz */api*. Wenn man die Applikation jetzt neu deployt, wird man von Tye gefragt, ob man den Ingress Controller installieren möchte. Diese Frage sollte man mit Ja beantworten. Dadurch verlängert sich das Deployment etwas und Tye quittiert einen Erfolg durch die Meldung der *IngressIP*:

```
Waiting for ingress to be deployed. This may take a long
  time.
    Retrieving details for ingress...
      IngressIP: '20.101.248.199'
```

Nun kann man im Browser diese IP eingeben und gelangt wieder zur eigenen Frontend-Applikation, und das ganz ohne Port-Forwarding.

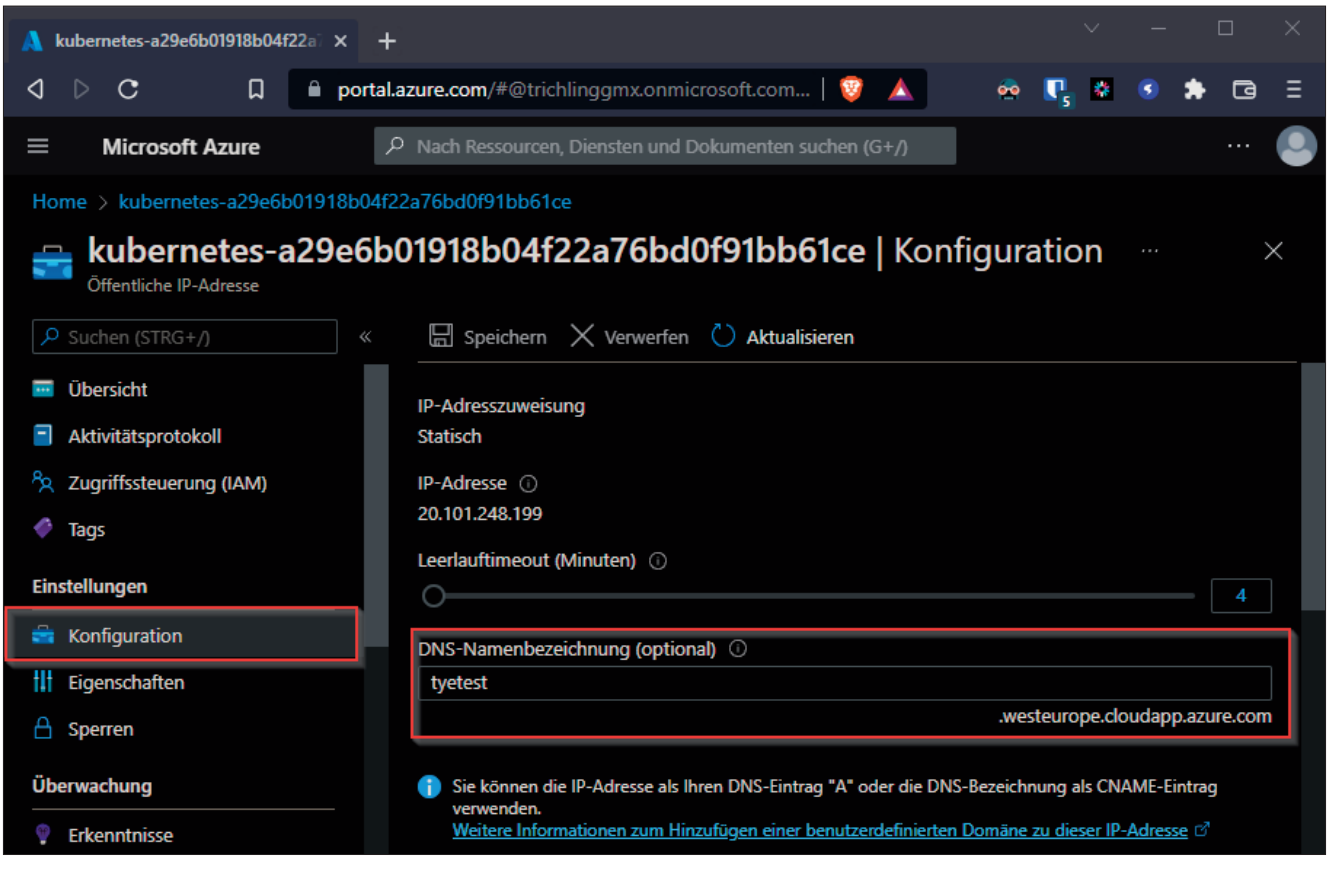
Die zusätzlichen Ingress-Regeln in den Zeilen 16 bis 18 und 27 bis 29 spezifizieren zusätzlich noch einen Hostnamen. Wenn man im Browser also nicht die IP-Adresse, sondern einen Hostnamen eingeben möchte, ist dieser in den Ingress-Regeln zu konfigurieren. Damit das mit unserem AKS-Cluster funktioniert, ist die öffentliche IP-Adresse entsprechend zu konfigurieren. Diese findet man in einer eigenen Ressourcengruppe, die zu jedem AKS-Cluster gehört. Für unseren Beispielcluster heißt diese *MC_tyetest_tyecluster_westeurope*. In dieser Ressourcengruppe befinden sich alle Infrastrukturobjekte des Clusters, unter anderem zwei öffentliche IPs. Die IP des Ingress Controllers ist mit dem Tag

```
service: ingress-nginx/ingress-nginx-controller
```

versehen. In der Konfiguration dieser IP-Adresse kann man einen DNS-Suffix eintragen. Dieser wird mit der Standard-Domain für Azure-Cloud-Apps verbunden (siehe Bild 3).

Verwendet man als DNS-Suffix *tyetest*, so ist die Anwendung danach unter

```
http://tyetest.westeurope.cloudapp.azure.com
```



DNS-Konfiguration für den Ingress Controller (Bild 3)

Listing 2: Auszüge aus der tye.yaml für RateMyBeer

```
 1:  name: lab.ratemybeer
 2:  services:
 3:  - name: sqlserver
 4:    image: mcr.microsoft.com/mssql/
         server:2019-latest
 5:    bindings:
 6:      - port: 1433
 7:
 8:    env:
 9:      - name: ACCEPT_EULA
10:       value: Y
11:     - name: SA_PASSWORD
12:       value: 1stChangeIt!
13:
14:   volumes:
15:     - source: ./Lab.RateMyBeer.SQLServer/Volumes
16:       target: /var/opt/mssql/data
17:
18: - name: lab-ratemybeer-checkins
19:   project: Lab.RateMyBeer.Checkins/
        Lab.RateMyBeer.Checkins.csproj
20:
21:   env:
22:     - name: ASPNETCORE_ENVIRONMENT
23:       value: Development
24:     - name: ASPNETCORE_URLS
25:       value: https://+:443;http://+:80
26:     - name: Dependencies__NServiceBus
            __TransportConnectionString
27:       value: host=localhost;username=guest;
            password=guest
28:     - name: ConnectionStrings
            __CheckinsDbConnectionString
29:       value: Server=localhost;
            Database=CheckinsDb;User=sa;
            Password=1stChangeIt!;
            MultipleActiveResultSets=true
30:
31:   bindings:
32:     - name: http
33:       containerPort: 80
34:       protocol: http
35:       port: 6002
36:     - name: https
37:       protocol: https
38:       containerPort: 443
39:       port: 6003
40:
41: - name: rabbitmq
42:   ...
43:
44: - name: lab-ratemybeer-frontend
45:   ...
46:
47: - name: lab-ratemybeer-frontend-api
48:   ...
```

erreichbar. Wenn man diese Konfiguration vorgenommen hat, die man über das Azure CLI auch skripten kann, braucht man die Ingress-Regeln ohne die Host-Angabe eigentlich gar nicht mehr.

Um die Anwendung über HTTPS verfügbar zu machen, ist noch der Cert-Manager einzubinden, um von Let's Encrypt Zertifikate zu beziehen. Wie das vor sich geht, wird in [2] beschrieben.

Das Backend kann man betrachten unter

```
http://tyetest.westeurope.cloudapp.azure.com/api/swagger
```

Für die MVC-Applikation im Beispiel wäre es gar nicht erforderlich, das API per Ingress für die Außenwelt verfügbar zu machen, da die Controller den Aufruf ja bereits innerhalb des Clusters erledigen. Man kann die Ingress-Definition für das Backend ab Zeile 22 also einfach löschen, und die Anwendung würde weiterhin problemlos funktionieren. Falls man für das Frontend jedoch auf ein SPA setzen möchte, ist neben dem Webserver, der die Index-Datei und das zugehörige JavaScript ausliefert, auch das Backend öffentlich zugänglich zu machen. Dies lässt sich dann wie in Listing 1 beschrieben erreichen.

Jenseits der grünen Wiese

Für ein neues Projekt, bei dem man eventuell mit Kubernetes starten möchte, ist Tye sicher eine sehr gute Wahl. Eine Applikation mit mehreren Diensten lässt sich so mit wenig Aufwand in den Cluster bringen. Doch wie sieht es mit bestehenden Projekten aus oder mit Projekten, die zum Beispiel über mehrere Git-Repositories oder Solutions verteilt sind?

Zur Klärung der ersten Frage ziehen wir das RateMyBeer-Repository [6] heran, das bereits in [2] als Beispiel gedient hat. Erst mal gehen wir also hemdsärmelig an die Sache heran: Repo klonen und *tye run*. Auf den ersten Blick sieht das auch gar nicht schlecht aus. Tye erkennt die *docker-compose.yml*-Datei und verwendet diese zur Konfiguration der Anwendung. Es werden Images gebaut und Container gestartet. Ein Blick auf das Dashboard bringt leider die erste Ernüchterung: Tye erkennt keine Bindings für die Dienste, die es gestartet hat. Es hat den Anschein, dass die *docker-compose.override.yml* nicht mit in Betracht gezogen wird. Das manuelle Kombinieren beider Dateien führt leider auch nicht zum Erfolg.

Da hilft nur das manuelle Erstellen einer *tye.yaml* aus den vorhandenen Compose-Dateien. Das gelingt recht einfach, denn das Dateiformat ist gut beschrieben [7] und wirkt auch etwas zugänglicher als das Docker-Compose-Format. Aus-

zugsweise ist diese in Listing 2 zu finden. Es fällt auf, dass die Bindings manuell angegeben werden müssen, damit Tye diese erkennt. Das Setzen von Umgebungsvariablen und das Einbinden von Volumes gelingt ohne Weiteres. Schade ist allerdings, dass Tye ein anderes Netzwerk aufbaut als Docker Compose. Während Letzteres alle Dienste unter ihrem jeweiligen Namen verfügbar macht, sind in Tye alle Dienste unter *localhost* zusammen mit dem entsprechenden Port anzugeben, zu sehen in den Zeilen 26 bis 29 für den SQL Server und den RabbitMQ Server. Das ist ärgerlich, weil auf diese Weise ein Deployment in den Kubernetes-Cluster nicht funktioniert, da die Services dort unter dem jeweiligen Servicenamen zu erreichen sind.

Die Lösung wäre, die Service-Discovery-Funktionalität von Tye zu nutzen. In diesem Fall bräuchte man die Umgebungsvariablen gar nicht explizit zu setzen, sondern würde Tye diese Arbeit erledigen lassen. Dafür sind allerdings Umbauten am Code erforderlich, was bei umfangreicheren Projekten schon einen erheblichen Aufwand bedeuten kann.

Was genau ist also zu tun? Für die Dienste, die einen Connection-String generieren sollen, ergänzt man im Service-Binding die Property *connectionString*, in der man ein Template für den Connection-String anlegen kann, das Platzhalter für Host und Port beinhalten kann (vergleiche Listing 3, Zeile 7). Auch der Bezug auf Umgebungsvariablen ist möglich. Bei den abhängigen Diensten entfernt man die Umgebungsvariablen, die zum Setzen der Connection-Strings oder der URIs für die konsumierten Dienste verwendet wurden (vergleiche Listing 3, Zeilen 22 bis 26). Im Beispiel des Checkin-APIs sind dies der Connection-String für den SQL Server und für RabbitMQ (vergleiche Listing 2, Zeilen 26 bis 29). Nach der Installation des NuGet-Pakets für die Konfiguration generiert man den Connection-String zur Datenbank nicht mehr so:

```
var checkinDbConnectionString = Configuration
  .GetConnectionString("CheckinsDbConnectionString");
```

sondern so:

```
var checkinDbConnectionString =
  string.Format(Configuration.GetConnectionString(
  "sqlserver"), "CheckinsDb");
```

Man beachte, dass hier der Connection-String unter dem Namen des Dienstes abgerufen wird. Außerdem ist der eigene Platzhalter für den Namen der Datenbank manuell zu ersetzen. Ähnlich verhält es sich mit dem Connection-String für RabbitMQ, der vor der Änderung so abgerufen wurde:

```
var transportConnectionString = context.Configuration[
  "Dependencies:NServiceBus:TransportConnectionString"];
```

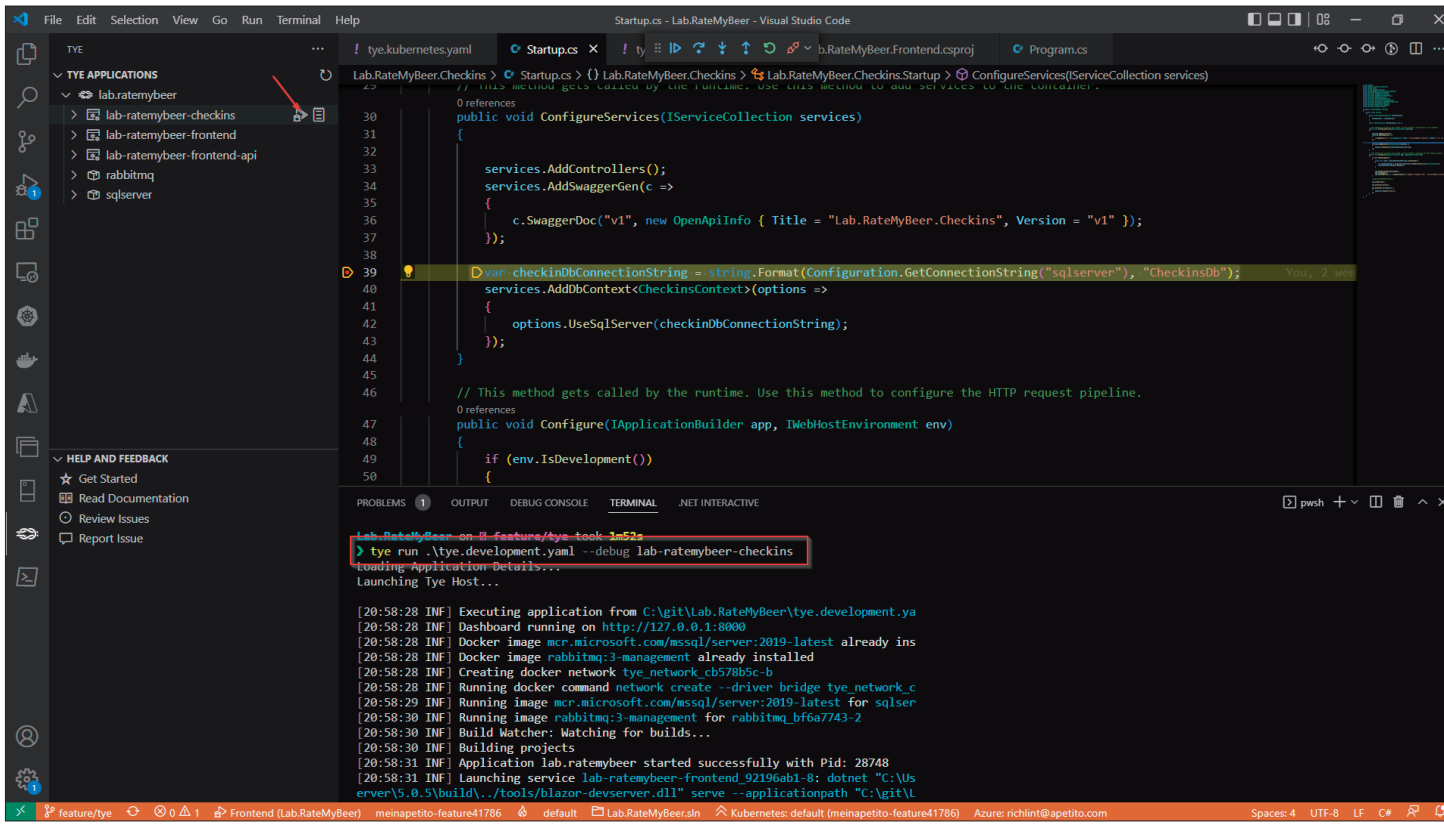
Unter Verwendung der Service Discovery wird daraus

```
var transportConnectionString = context.Configuration
  .GetConnectionString("rabbitmq", "amqp");
```

▶

Listing 3: Die für Service Discovery angepasste tye.yaml (Auszug)

```
 1: name: lab.ratemybeer
 2: services:
 3: - name: sqlserver
 4:   image: mcr.microsoft.com/mssql/
        server:2019-latest
 5:   bindings:
 6:     - port: 1433
 7:       connectionString: Server=${host};
            Database={0};User=sa;Password=${env:
            SA_PASSWORD};MultipleActiveResultSets=true
 8:
 9:   env:
10:     - name: ACCEPT_EULA
11:       value: Y
12:     - name: SA_PASSWORD
13:       value: 1stChangeIt!
14:
15:   volumes:
16:     - source: ./Lab.RateMyBeer.SQLServer/Volumes
17:       target: /var/opt/mssql/data
18:
19: - name: lab-ratemybeer-checkins
20:   project: Lab.RateMyBeer.Checkins/
          Lab.RateMyBeer.Checkins.csproj
21:
22:   env:
23:     - name: ASPNETCORE_ENVIRONMENT
24:       value: Development
25:     - name: ASPNETCORE_URLS
26:       value: https://+:443;http://+:80
27:
28:   bindings:
29:     - name: http
30:       containerPort: 80
31:       protocol: http
32:       port: 6002
33:     - name: https
34:       protocol: https
35:       containerPort: 443
36:       port: 6003
37:
38: - name: rabbitmq
39:
40: - name: lab-ratemybeer-frontend
41:
42: - name: lab-ratemybeer-frontend-api
```



Debugging von Tye-Projekten (Bild 4)

Auch hier greift man über den Dienstnamen auf den Connection-String zu. Da der RabbitMQ-Dienst zwei Bindungen hat, eine für das Web UI und eine für die Client-Verbindungen, ist der Name des Bindings hier als zweiter Parameter mit zu übergeben.

Auf den URL eines Dienstes, den man konsumieren möchte, würde man nicht mehr wie gehabt so zugreifen:

```
var checkinsApiBaseUrl = Configuration[
  "Dependencies:APIs:CheckinsApiBaseUrl"];
```

sondern über den Namen des gewünschten Dienstes und über die Erweiterungsmethode *GetServiceUri*:

```
var checkinsApiBaseUrl = Configuration.GetServiceUri(
  "lab-ratemybeer-checkins", "http");
```

Wenn man die Kröte geschluckt hat, dass ein paar Änderungen an der Applikation vorzunehmen sind, bekommt man diese dann auch unter Tye zum Laufen.

Von der Wiese auf den Dampfer

Was lokal geht, soll ja auch ganz einfach in Kubernetes laufen, nicht wahr? Und da wir ja gerade noch einen Cluster zur Hand haben, führen wir ganz selbstbewusst *tye deploy* aus – und werden enttäuscht. Die Builds der Docker-Images schlagen fehl. Der Grund dafür ist, dass die Dockerfiles neben den *csproj*-Dateien abgelegt sind, der Build-Context aber auf der Ebene der Solution eingestellt werden muss. Nur so hat man im Dockerfile Zugriff auf die anderen Projekte der Solution. Eine Suche in der Dokumentation fördert das Attribut *dockerFileContext* zutage, mit dem man den Build-Context setzen kann. Leider schafft die Verwendung des Attributs keine Abhilfe für das Problem.

Der Grund ist, das Tye verschiedene Arten von Services kennt, darin liegt eine seiner Stärken gegenüber Docker

Listing 4: Verschiene Arten von Tye-Diensten

```
1:  name: lab.ratemybeer
2:  registry: tyetestcontainers.azurecr.io
3:
4:  services:
5:  - name: sqlserver
6:    image: mcr.microsoft.com/mssql/
        server:2019-latest
7:
8:  - name: lab-ratemybeer-frontend
9:    dockerFile: Lab.RateMyBeer.Frontend/Dockerfi-
le
10:   dockerFileContext: .
11:
12: - name: lab-ratemybeer-checkins
13:   project: Lab.RateMyBeer.Checkins/
        Lab.RateMyBeer.Checkins.csproj
```

Compose, das nur mit Containern umgehen kann. Tye kennt des Weiteren noch C#-Projekte und Dienste, die auf fertigen Images basieren. In Listing 4 ist für jeden dieser Typen ein Beispiel dargestellt. Das Schlüsselwort *image* weist auf ein vorhandenes Docker-Image hin. Der Parameter *dockerFile* in Verbindung mit *dockerFileContext* weist Tye an, ein Image für diesen Dienst beim Starten zu bauen. Das Schlüsselwort *project*, das bisher für den eigenen Code verwendet wurde, startet einfach das Projekt selbst.

In der Verwendung von Container-Diensten, bei denen die Images selbst gebaut werden, Stichwort *dockerFile*, liegt dann auch die Lösung für das Deployment nach Kubernetes. Man ändert einfach die eigenen Dienste so um, dass man nicht mehr mittels *project* auf die Projektdatei verweist. Danach funktioniert auch das Deployment. Wer die lokalen Projekte lieber direkt starten möchte, kann einfach eine zweite Tye-Konfigurationsdatei speziell für Kubernetes anlegen. Hier lassen sich dann auch die Ingress-Definitionen ablegen, die man für das lokale Starten nicht benötigt.

Dabei ist aber zu beachten, dass Tye die beiden Dienste mit den vorhandenen Images nicht automatisch in den Cluster deployt. Das ist leider von Hand zu erledigen.

Was ist, wenn was schiefläuft?

Leider läuft ja beim Entwickeln nicht immer alles so rund, wie der Entwickler sich das wünschen würde. Und mittels *tye run* kann man zwar eine Menge Projekte, Container und Images starten, debuggen kann man das aber leider nicht so gut. Das Dashboard liefert zwar schon gute Anhaltspunkte für die Fehlersuche, aber manchmal muss eben der Debugger ran.

Auch daran wurde gedacht, indem man *tye run* den Parameter *--debug* und den Namen eines Dienstes übergibt:

```
tye run .\tye.development.yaml --debug
  lab-ratemybeer-checkins
```

Dann wartet Tye mit dem Starten so lange, bis man sich über die Visual Studio Code Extension mit dem entsprechenden Projekt verbunden hat (vergleiche Bild 4)

Dazu klickt man nach dem Konsolenbefehl auf das *Attach*-Icon neben dem Dienstnamen, das erscheint, wenn man mit der Maus über den Dienstnamen fährt.

Fazit

Der Eindruck von Tye ist durchwachsen. Für Greenfield-Projekte, bei denen man mit Containern und Kubernetes starten möchte, macht es viele Dinge leichter. Wenn man die Service Discovery von vornherein verwendet, fallen viele Probleme weg, mit denen man sich sonst selbst herumärgern muss. Außerdem hat man den Zugriff auf Dienste dann auf eine konsistente Art gelöst. Sobald man später auf den Dampfer möchte, ist der Umstieg leicht zu schaffen. Selbst wenn das Projekt größer wird, kann Tye noch mitwachsen – es unterstützt nämlich auch Projekte, die sich über mehrere *Tye.yaml* und sogar über mehrere GIT-Repositories erstrecken [8].

Hier lauert aber bereits die erste Gefahr: Weil Tye die Kubernetes-Manifeste erstellt, entfällt hier eine Lernkurve, die einem bei einer Fehlersuche ziemlich böse auf die Füße fallen kann. Dass externe Dienste (Typ *image*) nicht automatisch in den Cluster deployt werden, ist schon mal ein Vorgeschmack auf das, was noch auf einen zukommen kann.

Ist man abseits der grünen Wiese unterwegs, fällt der Einstieg in Tye schon schwerer. Gerade der Umstieg von Docker Compose geht nicht leicht vonstatten. Hier hat man wirklich eine gute Gelegenheit versäumt. Gerade bei größeren Projekten stört es auch, dass Tye beim Deployment die Manifeste nur temporär erstellt und nicht als Dateien ablegt, sodass man diese in die Versionskontrolle bringen könnte.

Durch die Service Discovery werden die Manifeste auch recht länglich, da Tye jedes Mal die Informationen für alle Dienste dort ablegt, unabhängig davon, ob ein Dienst nun eine Abhängigkeit aufweist oder nicht. Auf der anderen Seite kann auch ein Brownfield-Projekt von einem sauberen Ansatz für die Service Discovery profitieren – auch oder gerade weil der Code dafür wahrscheinlich ein wenig refaktoriert werden muss. Hier ist der Ansatz von Tye dann wiederum erfreulich niedrigschwellig.

Anlass zur Vorsicht gibt allerdings die Aktivität von Tye. Seit der ersten Erwähnung in einem Blogbeitrag von 2020 sind zwar einige Releases erschienen, das letzte sogar noch am 15. Februar 2022. Trotzdem macht das Repository von Tye keinen besonders lebendigen Eindruck. Was schade ist, denn die Ansätze in dem Tool sind wirklich gut.

Insgesamt kann Tye aber auch vor diesem Hintergrund für die Beschleunigung der lokalen Entwicklung und in der Bootstrapping-Phase eines Projekts eine gute Starthilfe sein. Wenn ein Projekt reift und größer wird, kann man auf den Choke dann irgendwann verzichten. Sollte sich Tye dann nicht mehr weiterentwickeln, kann man mit eigenem Schub weiterfliegen. ■

[1] Tobias Richling, Ab in den Container, dotnetpro 9/2021, Seite 8 ff., www.dotnetpro.de/A2109WebDocker
[2] Tobias Richling, Ab auf den Dampfer, dotnetpro 9/2021, Seite 16 ff., www.dotnetpro.de/A2109Kubernetes
[3] Tye, www.dotnetpro.de/SL2208Tye1
[4] Tye-Quickstart-Tutorial, www.dotnetpro.de/SL2208Tye2
[5] Kubernetes-Tools für Visual Studio Code, www.dotnetpro.de/SL2208Tye3
[6] RateMyBeer-Repository, www.dotnetpro.de/SL2208Tye4
[7] tye.yaml-Schema, www.dotnetpro.de/SL2208Tye5
[8] Using tye with Multiple Repositories, www.dotnetpro.de/SL2208Tye6

Tobias Richling
betreut als Softwareentwickler die B2B-Plattformen der apetito AG. Er sucht stets nach neuen Möglichkeiten zur Verbesserung der Funktionalität und Performance. Sie erreichen ihn unter tobias.richling@apetito.de oder per **@trichling**.

dnpCode A2208Tye