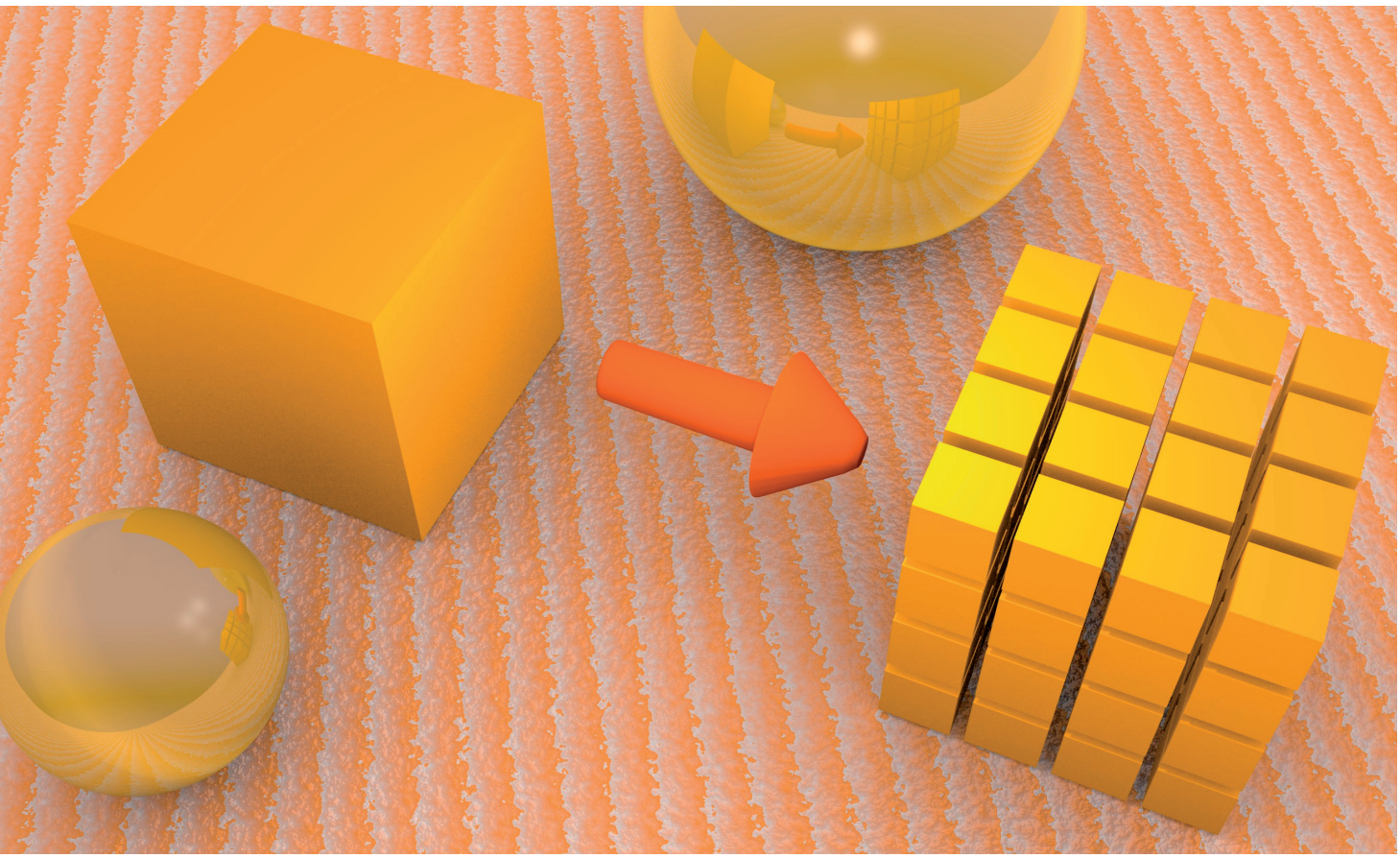


SERVICE HOST, TEIL 1

HTTP-Services für jedermann

Alle reden von Microservices und serviceorientierten Architekturen – aber wie geht das?

Wenn Sie heute an einer zehn Jahre alten Desktop-Anwendung mit einem Oracle-Server im Backend schrauben, hört sich der Begriff Microservices für Sie wohl ziemlich befremdlich an. Ihre Kompetenzen liegen bei allem Möglichen, nur eher nicht beim Aufsetzen von „irgendetwas mit Internet". Aber auch wenn Ihr Produkt schon mit ASP.NET MVC im Web läuft, heißt das nicht, dass Sie serviceorientiert entwickeln. Monolithische Anwendungen gibt es auch im Internet.

Ich bin auch eher der Desktop-Entwickler. Mit Code im Web habe ich mich bisher schwergetan. Allemal, wenn es ums Deployment geht. Doch irgendwann wollte ich es wissen. Wenn alle Welt über Microservices redet, dann will ich damit auch experimentieren – aber möglichst einfach. Ich will mich nicht mit Infrastruktur aufhalten, sondern schnell an den Punkt der Nutzung kommen. Ich möchte sehen, wie Services zusammenspielen können, wie ich Funktionalität auf Services verteilen kann. Einen Service zu schreiben und zu deployen darf mir dabei nicht im Wege stehen.

So habe ich begonnen, genau das für mich zu vereinfachen. Und nun will ich Ihnen meine Lösung auch vorstellen. Aber zuerst etwas Grundlage, damit Sie verstehen, was ich überhaupt mit Service und Serviceorientierung meine.

Was ist ein Service?

Für mich gehört ein Service zur selben Kategorie wie eine Klasse. Beides sind Module. Module dienen der Sammlung von Logik, die demselben Zweck dient. Logik umfasst alle programmiersprachlichen Anweisungen, die Verhalten herstellen. Das sind Transformationen, Kontrollstrukturen und API-Aufrufe (zum Beispiel I/O). Ein Beispiel zeigt Bild 1.

Logik an sich ist allerdings „bedeutungslos", das heißt, dass sie keine Aussage darüber trifft, was sie tut. Das müssen

```
Console.Write("Enter text: ");
var text = Console.ReadLine();

var chunks = text.Split(new[] {' ', '\n', '\r', '\t'},
                        StringSplitOptions.RemoveEmptyEntries);
var words = chunks.Where(ch => char.IsLetter(ch[0])).ToArray();
var n = words.Length;

Console.WriteLine($"Number of words: {n}");
```

Ein Beispiel für pure Logik (Bild 1)

```
[TestFixture]
public class Wordcounting_tests
{
    [Test]
    public void Count()
    {
        var result = Wordcounting.Count("a b2\n1c   d,\tefg");
        Assert.AreEqual(4, result);
    }
```

Tests als semantischer Kontrakt von Modulen (Bild 4)

```
public static int Count(string text) {
    var chunks = text.Split(new[] {' ', '\n', '\r', '\t'},
                            StringSplitOptions.RemoveEmptyEntries);
    var words = chunks.Where(ch => char.IsLetter(ch[0])).ToArray();
    return words.Length;
}

Console.Write("Enter text: ");
var text = Console.ReadLine();

var n = Count(text);

Console.WriteLine($"Number of words: {n}");
```

Funktionen geben der Logik Bedeutung (Bild 2)

Sie durch Interpretation herausfinden. Kein geringer Aufwand, wie Sie schon bitter erfahren haben, wenn Sie Code Ihrer Kollegen verändern mussten. Bedeutung bekommt Logik, indem wir sie in Funktionen wickeln, die klarmachen, auf welchen Daten die Logik arbeitet (Input), welche Daten sie erzeugt (Output) und was das Ganze eigentlich soll.

In Bild 2 habe ich einen Teil der Logik aus Bild 1 in dieser Weise verpackt. Am Aufrufort ist nun klarer, was da eigentlich passiert, ohne dass dazu Logik interpretiert werden müsste. Unter einem Funktionsnamen wird also Logik nach ihrer Tätigkeit zusammengefasst, hier ist dies das Zählen der Wörter in einem Text.

Module nun fassen verschiedene Tätigkeiten unter dem Dach eines übergreifenden Zwecks zusammen. Module aggregieren Funktionen, die sich in gewisser Hinsicht ähneln. Ein Beispiel zeigt Bild 3. Die Klasse *Wordcounting{}* steht für den Zweck „Alles, was mit dem Zählen von Wörtern zu tun hat“ und versammelt in sich Funktionen, die dazu beitragen.

Nach außen veröffentlicht wird nur eine von diesen Funktionen. Sie wird für Interessenten an diesem Zweck als relevant erachtet. Die anderen Funktionen sind Details, die zur Erfüllung des Zwecks nötig sind. Ob es sie aber überhaupt gibt und wie die Aspekte des Zwecks – zum Beispiel die Bestimmung, was überhaupt ein Wort ist – erfüllt werden, ist für Nutzer der Klasse irrelevant. Diese Details können sich jederzeit ändern, solange der semantische Kontrakt der Klasse, das heißt ihr Zweck, weiterhin erfüllt wird. Um das sicherzustellen, sind automatisierte Tests geeignet (vergleiche Bild 4).

```
public static class Wordcounting
{
    public static int Count(string text) {
        var chunks = Split_into_word_candidates(text);
        var words = Discard_non_words(chunks);
        return words.Count();
    }

    private static IEnumerable<string> Split_into_word_candidates(string text)
        => text.Split(new[] {' ', '\n', '\r', '\t'}, StringSplitOptions.RemoveEmptyEntries);

    private static IEnumerable<string> Discard_non_words(IEnumerable<string> chunks)
        => chunks.Where(ch => char.IsLetter(ch[0])).ToArray();
}
```

Eine Klasse aggregiert Funktionen mit dem gleichen Zweck (Bild 3)

Der semantische Kontrakt einer Klasse ist die Summe der Leistungen der Logik in ihren Funktionen. Der syntaktische Kontrakt hingegen ist die Summe der Signaturen der öffentlichen Funktionen.

Bei der Klasse in den bisherigen Beispielen besteht der syntaktische Kontrakt nur aus einer Funktion und ist implizit. Der syntaktische Kontrakt ist nur an der Klasse selbst abzulesen.

Mit einem Interface können Sie syntaktische Kontrakte jedoch von Klassen ablösen. Es kann dann potenziell mehrere Implementationen desselben Kontrakts geben. Und Nutzer des Kontrakts müssen nicht mehr wissen, welche konkrete Implementation ihnen vorliegt (Dependency Inversion Principle (DIP), Bild 5).

Und jetzt Services

Services sind – wie eingangs formuliert – ebenfalls Module. Größere, gröbere, umfassendere als Klassen, aber ebenfalls Module. Services sind also nichts Neues oder gar Mythisches, sondern sie sind handfest und sogar uralt. Nur leider hat man das jahrzehntelang nicht realisiert. Man war so fixiert auf andere Dinge, dass Services als Module keine Rolle spielten.

Dabei sind Services so einfach. Finde ich jedenfalls. Services sind für mich lediglich Module mit einem separaten, expliziten Kontrakt, der plattformneutral ist. Das ist alles. Aber daraus folgt einiges.

Erstens folgt daraus, dass der Kontrakt eines Service kein Interface wie oben sein kann. Ein Interface ist plattformspezifisch. Das obige Interface kann nicht von einer Java-Klasse implementiert werden. Auch dann nicht, wenn es in einer eigenständigen Kontrakt-Assembly vorliegen sollte.

Service-Kontrakte müssen also in anderer Form definiert sein. Das kann eine formale IDL (Interface Description Language) sein. Oder aber auch ein schlichter Zettel (Bild 6).

Zweitens folgt daraus, dass Services in einem eigenen Betriebssystemprozess bereitgestellt werden müssen. Denn da ihre Kontrakte plattformneutral sind, können sie ja in Java oder Python oder C# implementiert werden. ▶

Solche Implementationen lassen sich jedoch nicht in einem Prozess mischen. Also bekommt jeder Service einen eigenen.

Sie sehen, Serviceorientierung hat vom Prinzip her erst einmal gar nichts mit spezieller Infrastruktur zu tun. Sie müssen nicht Docker- oder Kubernetes-Spezialist sein, um serviceorientiert zu entwickeln. Der wesentliche Schritt besteht darin, Module jenseits von Klassen als eigenständige Prozesse zu denken. Schon dadurch gewinnen Sie einige Vorteile:

- Services können in unterschiedlichen Sprachen entwickelt werden. So können Sie die Vorteile von Plattformen ausreizen oder Menschen am Projekt beteiligen, die sonst nicht erreichbar wären.
- Services stellen deutlich getrennte Zwecke dar, an denen leicht verteilt gearbeitet werden kann. Die expliziten Kontrakte ziehen eine stabile, undurchlässige Grenze für die Arbeitsorganisation.

Insofern war schon Unix serviceorientiert. Der Kontrakt zwischen den vielen Kommandozeilenwerkzeugen bestand darin, Input und Output über *stdin* und *stdout* fließen zu lassen. Als Beispiel der Einsatz von drei Services auf meinem Mac:

```
<span class="text-inlinelisting">ps aex |
  grep "/Applications/" | wc
</span>
```

Die Zwecke der einzelnen Services sind:

- Prozessübersicht (*ps*)
- Textfilterung mit regulären Ausdrücken (*grep*) und
- Textstatistik (*wc*)

Im Verbund wie oben arbeiten diese Services zusammen, um die Zahl der Prozesse zu bestimmen, die auf installierten Anwendungen basieren. Von knapp 350 Prozessen sind das derzeit bei mir 61.

Bei den Unix-Services ist der Kontrakt sehr simpel und immer gleich. Deshalb gibt es auch eine so große Zahl von Unix-Services, die immer und immer wieder zu unterschiedlichem Verhalten rekombiniert wurden und werden. Diese Möglichkeit der Service Composition mit den Operatoren |, >, >> war geradezu der USP von Unix und allen Derivaten. Jeder konnte mitmachen, entweder durch ausgefeilte Kombination der Services (Integration) oder durch Entwicklung neuer Services (Aggregation von Logik hinter einem Kontrakt).

Daran möchte ich mit meiner Servicedefinition anschließen. Deshalb zeigt Bild 6 einen Service-Kontrakt, auch wenn der anders aussieht als der der Unix-Services. Das Prinzip ist dasselbe: Kontrakt plattformneutral, Service als eigenes Programm.

Wie ein Service angesprochen wird, wie er Ergebnisse liefert, das ist egal. Alles ist erlaubt – solange es mit verschiedenen Programmiersprachen beziehungsweise auf verschiedenen Entwicklungsplattformen implementiert werden kann. Input via *stdin* oder als Kommandozeilenargumente? Egal. Output über *stdout* oder in einer Datei? Egal. Ebenso egal: Input über HTTP als JSON und auch Output über HTTP als JSON zurück.

Services mit einer REST-Schnittstelle sind insofern für mich nur Sonderfälle. HTTP/JSON stellt nur eine von vielen Möglichkeiten plattformneutraler Kommunikation dar.

HTTP-Services

Zugegeben, HTTP-Services mit einer REST-Schnittstelle sind derzeit der Hit. Und HTTP-Services waren auch das, was mich zu meinen Anstrengungen veranlasst hat, Serviceorientierung einfacher zu machen.

Console-Services wie für Unix zu schreiben und zu deployen ist trivial. „Wahre“ Serviceorientierung aber betreibt Services im Netz, oder? Im Intranet oder Internet stehen Services bereit, um von allen Seiten aus angerufen zu werden. Das wollte ich so einfach machen wie Console-Services. Und wenn nicht für den endgültigen Produktionsbetrieb, dann doch zumindest für den soliden prototypischen Betrieb.

HTTP als Protokoll mit beigeordneten Standards ist verwirrend umfangreich. Mein Lösungsvorschlag für *simple services* ist deshalb nicht vollständig oder universell. Doch das aus meiner Sicht erst mal Wichtigste geht. Hier die Charakteristika von HTTP-Services, die ich berücksichtigt habe:

- HTTP-Services werden über verschiedene Routes angesprochen, zum Beispiel *GET /wordcount* oder *PUSH /texts* oder *DELETE /projects/1234*. Zur Route gehören das HTTP-Verb und ein URL-Pfad.
- Routes können einen variablen Anteil haben (Platzhalter), zum Beispiel eine ID wie in *GET /api/v1/projects/{id}*, der beim Aufruf konkretisiert werden muss, zum Beispiel *GET /api/v1/projects/9876abcd*.
- Routes können mit Parametern versehen werden (Querystring Parameters), zum Beispiel *GET /wordcount?text=hello%20world*.

```
public interface ICounting {
    int Count(string text);
}

public class Wordcounting : ICounting
{
    public int Count(string text) {...}

    private static IEnumerable<string> Split_into_
        => text.Split(new[] {' ', '\n', '\r', '\t'

    private static IEnumerable<string> Discard_non
        => chunks.Where(ch => char.IsLetter(ch[0]))
}

public class Charcounting : ICounting {
    public int Count(string text) {
        return text.Count(char.IsLetterOrDigit);
    }
}
```

```
public class App
{
    private readonly ICounting _counting;

    public App(ICounting counting) {
        _counting = counting;
    }

    public void Run() {
        Console.Write("Enter text: ");
        var text = Console.ReadLine();

        var n = _counting.Count(text);

        Console.WriteLine($"Counted: {n}");
    }
}
```

Ein Interface als expliziter syntaktischer Kontrakt (Bild 5)

- Serviceaufrufe können eine Payload eines gewissen Inhaltstyps haben, zum Beispiel einen JSON-String.
- Serviceaufrufe liefern einen HTTP-Result-Code, zum Beispiel 200 im Erfolgsfall.
- Serviceaufrufe liefern ein Resultat als Payload mit einem gewissen Inhaltstyp, zum Beispiel einen JSON-String.

Idealerweise soll für die Bereitstellung einer Funktionalität als Service nichts an ihr verändert werden müssen. Und das, was zusätzlich zu tun ist, damit sie über HTTP ansprechbar ist, soll minimal sein.

Ob ein Service ein REST-Service ist oder nicht, ist mir dabei nicht so wichtig. Sie müssen sich Routes für Ihre Szenarien überlegen. Die können Ressourcen benennen oder nicht. Die können versioniert sein oder nicht. Was Sie bereitstellen, ist in jedem Fall ein HTTP-Service.

HTTP-Services mit dem Service Host

Um HTTP-Services schnell und einfach aufsetzen zu können, habe ich für mich den Service Host entwickelt. Sie finden ihn bei GitHub als Open-Source-Projekt [1] und bei NuGet als Paket zum sofortigen Einsatz [2].

Der Service Host besteht aus zwei Teilen:

- Der Kontrakt definiert Attribute, mit denen Sie Serviceklassen als solche kennzeichnen.
- Der Host baut einen Server auf und bietet Serviceklassen zum Aufruf via HTTP an.

Bei Start sucht der Service Host in allen DLLs des Arbeitsverzeichnisses nach Serviceklassen. Sie müssen also nichts manuell registrieren. Darüber hinaus können Sie ihm aber auch explizit Serviceklassen mitgeben.

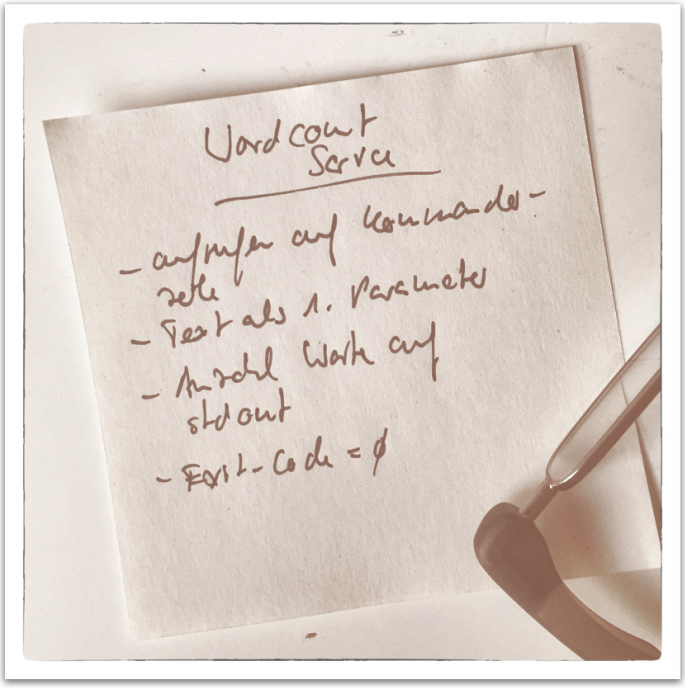
Der Service Host kann über das im Paket enthaltene Programm *servicehost.exe* gestartet werden. Sie müssen also außer Ihrer Serviceklasse keinen Code schreiben. Oder Sie übernehmen mehr Kontrolle und starten den Service Host aus Ihrem eigenen Programm.

Beim Start nennen Sie dem Service Host den URI, auf der er lauschen soll, als Kommandozeilenparameter, zum Beispiel *servicehost.exe http://localhost:9000*. Wenn Sie das nicht tun, sucht *servicehost.exe* nach dem URI in der Umgebungsvariablen *SERVICEHOST_ENDPOINT*.

Als HTTP-Server basiert Service Host auf dem leichtgewichtigen Server-Framework NancyFx [3]. Diesen Server benutze ich schon lange, wann immer ich kann, um „irgendetwas mit Web“ zu machen. IIS sind mir ein Graus; außerdem arbeite ich auf einem Mac und ansonsten mit Linux-Servern. Aber selbst nginx [4] ist mir zu groß. Was ich will: Überschaubarkeit, Leichtigkeit, Eingriffsmöglichkeit bei Bedarf, und Self-Hosting. Das alles bietet NancyFx.

Aber so schön NancyFx auch ist, es ist für „den kleinen Service zwischendurch“ immer noch zu viel lästiger Aufwand. Zu viel Wiring ist vorzunehmen, um Funktionalität zu hosten. Ich muss mir zu viel an NancyFx-Eigenheiten merken. Das ist mir zu aufwendig, wenn ich nur gelegentlich einmal wieder mit Services experimentieren will.

Mit dem Service Host habe ich deshalb NancyFx so gekapselt, dass das Hosting auf null Zeilen Code zusammenschnurrt, wenn man *servicehost.exe* aufruft, oder auch mit einer Zeile Code im eigenen Programm abgehandelt ist:



Papier-Version eines Service-Kontrakts (Bild 6)

```
ServiceHost.Run(new Uri("http://localhost:808"));
```

Der Rest geschieht dann wie von Zauberhand allein.

Ausblick

So weit zur Theorie. Die Umsetzung in die Praxis beschreibt der zweite Teil dieses Artikels in der nächsten dotnetpro-Ausgabe. Sie erfahren dort, was zu tun ist, um das vorhandene Wortzählungsmodul als HTTP-Service anzubieten. Und das zeigt Ihnen, dass HTTP-Services mit dem Service Host so einfach werden, dass ein ganz anderes, freieres Herangehen an die Softwarearchitektur möglich wird. ■

[1] Der Service Host bei GitHub, https://github.com/ralfw/servicehost
[2] Der Service Host als NuGet-Paket, https://www.nuget.org/packages/servicehost
[3] NancyFx, http://nancyfx.org
[4] nginx, https://nginx.org

Ralf Westphal
ist freiberuflicher Berater, Referent, Autor und Trainer für Themen rund um Softwarearchitektur und die Organisation von Softwareteams. Er ist Mitgründer von Clean Code Developer (CCD) und propagiert kontinuierliches Lernen.
info@ralfw.de

dnpCode A1806Servicehost