Zwölf Regeln für Web- und Cloud-Anwendungen

Originalartikel

Backup

<html> <time datetime=„2020-11-17T13:51:00+01:00“>17.11.2020 13:51</time>Golo Roden Das Entwickeln skalierbarer und verlässlicher Anwendungen für Web und Cloud ist ein komplexes Thema, das eine gewisse Erfahrung fordert. Dennoch gibt es Leitplanken für den einfachen Einstieg, allen voran die Regeln der 12-Factor-Apps. Was hat es mit damit auf sich? Für die Entwicklung von Web- und Cloud-Anwendungen gibt es verschiedene Architekturtypen. Doch je größer und komplexer eine Anwendung wird, desto eher werden Aspekte wie Skalierbarkeit, Hochverfügbarkeit und Ausfallsicherheit relevant. Diese Anforderungen lassen sich allerdings nicht mit einer monolithischen Architektur umsetzen, weshalb man bei solchen Anwendungen häufig auf ein Netz aus verteilten Diensten stößt, die einander ergänzen. Damit ein derartiges Netz funktionieren kann, sind an die einzelnen Dienste gewisse Anforderungen zu stellen. Beispielsweise ist für eine elastische Skalierbarkeit unabdingbar, dass sich Dienste je nach Bedarf starten und auch beenden lassen, ohne dafür in größerem Rahmen Zeit einplanen zu müssen. Das wiederum bedingt gewisse Herangehensweisen an die Entwicklung solcher Dienste. Da verteilte Architektur an sich kein einfaches Thema ist, fällt auch der Einstieg häufig schwer. Um diesem Problem entgegenzuwirken und einen einfachen Einstieg in die Thematik zu ermöglichen, hat der Cloud-Plattform-Anbieter Heroku schon vor etlichen Jahren zwölf Regeln verfasst, die als Leitplanken für den Entwurf und die Entwicklung von Diensten gelten können. Selbstverständlich umfasst verteilte Architektur weitaus mehr als diese zwölf Regeln, aber sie ermöglichen einen strukturierten erstem Kontakt mit dem Thema. Geführt werden diese Regeln unter dem Begriff der 12-Factor-Apps. Im folgenden werden diese Regeln vorgestellt, allerdings nicht in ihrer eigentlichen Reihenfolge, sondern thematisch gruppiert. Einige der Regeln liegen inhaltlich nämlich näher beieinander als andere, weshalb es sinnvoll ist, sie als einen gemeinsamen Block zu behandeln. Die erste Regel besagt, dass es für eine Anwendung genau eine Codebase geben sollte. Das bedeutet, dass der Code für eine Anwendung in einem einzigen Repository und nicht über zahlreiche Repositories verstreut vorliegen sollte. Das mag trivial und offensichtlich erscheinen, gehört aber tatsächlich nicht in allen Projekten zum Alltag. Selbstverständlich verbietet das nicht, in mehreren Anwendungen gleichsam genutzten Code in ein gemeinsames Modul zu extrahieren und dieses in ein eigenes Repository auszulagern. Vielmehr ist damit gemeint, dass jener Code, der tatsächlich anwendungsspezifisch ist, an einem einzigen Ort abgelegt sein sollte, der nur für diese Anwendung gedacht ist. Aus dieser Codebase repräsentiert jeder einzelne Commit praktisch eine eigenständige Version, die deployt werden könnte. Das mag nicht für jeden Commit sinnvoll sein, aber jede Änderung am Code lässt sich einzeln adressieren und könnte als Grundlage für ein Deployment dienen. Das bedeutet, dass es von einer Codebase unterschiedliche Versionen und von jeder dieser Versionen wiederum auch mehrere Deployments geben kann. Die zweite Regel besagt, dass man Codefragmente, die in verschiedenen Anwendungen verwendet werden oder die anwendungsunabhängig und damit generisch sind, in eigene Module extrahieren sollte, die in jeweils einem eigenen Repository verwaltet werden. Diese Module können dann als Abhängigkeiten in die Anwendungen eingebracht werden, sollten dort aber explizit deklariert werden. Das bedeutet, dass man nicht davon ausgehen sollte, dass eine bestimmte Abhängigkeit in einer bestimmten Version systemweit

vorhanden ist, sondern die benötigten Abhängigkeiten explizit in den Kontext der Anwendung installieren muss. Dazu benötigt man im Idealfall eine Paketverwaltung, die das Verwalten von Abhängigkeiten und deren Versionen vereinfacht. Welche Paketverwaltung das konkret ist, hängt von der verwendeten Technologie ab – für Node.js wäre das beispielsweise npm in Verbindung mit der Datei package.json. Eine Anwendung, die sich an die Regeln der 12-Factor-Apps hält, sollte also ihren Code und den Code aller Abhängigkeiten in einem Verzeichnis und dessen Unterverzeichnissen verwalten, aber nicht von Dateien oder Verzeichnissen außerhalb dieses Anwendungsverzeichnisses abhängen. <figure class=„video akwa-inline-video video–fullwidth“><figcaption class=„bildunterschrift akwa-caption“>Die Codebase als Single-Source of Truth</figcaption></figure> Die sechste Regel besagt, dass Anwendungen aus einem oder mehreren Prozessen bestehen sollen. Das heißt, dass, obwohl es nur eine Codebase geben darf, damit nicht zwingend gemeint ist, dass auch sämtlicher Code in einem einzigen Prozess ausgeführt wird. Das wäre im Hinblick auf die eingangs erwähnte verteilte Architektur, die viele verschiedene Dienste zu einem großen Ganzen kombiniert, auch nicht sinnvoll. Eine Anwendung in Prozesse zu zerlegen, bietet auch den Vorteil, dass diese individuell und unabhängig voneinander deployt, gestartet und aktualisiert werden können. Auch ein Ausfall eines Dienstes bedeutet nicht notwendigerweise den Ausfall der gesamten Anwendung: Im Idealfall läuft die Anwendung anstandslos weiter, zwar mit gegebenenfalls geringerer Performance oder mit dem Ausfall einiger Features, aber eben ohne vollständig auszufallen. Die achte Regel ergänzt diese Denkweise, indem sie vorgibt, dass man über Prozesse skalieren soll. Die Idee dahinter ist, dass man – spätestens wenn man eine Anwendung auf mehrere Server verteilt – ohnehin über Prozessgrenzen hinweg skalieren muss, weshalb man dieses Vorgehen von vornherein einplanen sollte. Das bedeutet auch, sich von Anfang an Gedanken über die Synchronisation von Prozessen und dementsprechend ebenfalls über Interprozesskommunikation (IPC) zu machen. Das Praktische daran ist, dass diese Mechanismen auch auf einem einzigen Server funktionieren, wenn man lediglich mehrere Instanzen der Anwendung auf einer Maschine betreiben will. Damit unterscheidet sich das Modell von der Skalierbarkeit mit Threads, und das Skalierungsmodell bleibt so oder so das gleiche. Die neunte Regel besagt, dass es essenziell ist, Prozesse zügig starten und stoppen zu können. Das zielt auf den eingangs bereits erwähnten Punkt ab, elastisch skalieren zu können: In dem Moment, zu dem mehr Performance benötigt wird, sollte es ein Leichtes sein, zusätzliche Instanzen zu starten. Es liegt auf der Hand, dass das dann keinen umfangreichen, aufwendigen und zeitintensiven Vorgang nach sich ziehen darf – ansonsten würde der gewünschte Effekt nämlich verpuffen. Ähnliches gilt für das Beenden eines Prozesses: Auch hier sollte ein Dienst so gebaut sein, dass es problemlos möglich ist, ihn im laufenden Betrieb abzuschießen, ohne dass zunächst noch Aufräumarbeiten erforderlich wären. Auf dem Weg wird es möglich, nicht mehr benötigte Ressourcen zügig auch wieder freizugeben. Wird diese Regel beachtet, ist es auch weitaus einfacher, Dienste in Docker beziehungsweise in der Cloud zu betreiben. Die zwölfte Regel besagt, dass administrative Aufgaben ebenfalls als Prozesse implementiert werden sollten, die auch Bestandteil der einen Codebase einer Anwendung sein dürfen. Auf dem Weg kann man einer Anwendung zum Beispiel Kommandozeilenwerkzeuge beilegen, die helfen, die Anwendung zu verwalten und zu steuern. <figure class=„video akwa-inline-video video–fullwidth“><figcaption class=„bildunterschrift akwa-caption“>Warum Prozesse so wichtig sind</figcaption></figure> Die siebte Regel besagt, dass eine Anwendung ihre eigene Funktionalität über Ports zur Verfügung stellen sollte – dass es also eine im Netzwerk zur Verfügung gestellte API geben sollte, auf die andere Anwendungen von außen zugreifen können. Der Grund für diesen Ansatz ist, dass er

unabhängig von Server- und Plattformgrenzen funktioniert, da sich auf Ports nicht nur lokal, sondern eben auch remote zugreifen lässt. Das wiederum ist die Grundlage für Skalierbarkeit. Die vierte Regel kehrt diesen Ansatz um und besagt, dass externe Dienste ebenfalls über Ports angebunden werden sollten. Auf dem Weg besteht kein Unterschied zwischen dem Bereitstellen und dem Konsumieren von Funktionalität, die Kommunikation erfolgt stets über Ports. Das gilt aber nicht nur für fachliche Dienste, sondern auch für Infrastrukturkomponenten wie Datenbanken oder Message-Queues. Zu beachten ist, dass beide Regeln nicht vorgeben, welches Protokoll zu verwenden ist. Das ist bewusst offen gelassen, sodass man HTTP, HTTPS, TCP, UDP oder ein beliebiges anderes Protokoll je nach Bedarf auswählen kann. <figure class=„video akwa-inline-video video–fullwidth“><figcaption class=„bildunterschrift akwa-caption“>Über Ports mit Diensten kommunizieren</figcaption></figure> Die dritte Regel betrifft die Art, wie eine Anwendung konfiguriert wird. Sie besagt, dass man Konfiguration mit Hilfe von Umgebungsvariablen vornehmen soll. Das bietet gleich mehrere Vorteile. Zum einen ist das Vorgehen plattformunabhängig, was beispielsweise für Kommandozeilenparameter und Konfigurationsdateien in der Regel nicht gilt, ganz zu schweigen von den zahlreichen Dateiformaten. Zum anderen läuft man nicht so schnell Gefahr, Umgebungsvariablen versehentlich in die Versionsverwaltung einzuchecken, da sie – anders als beispielsweise eine Konfigurationsdatei – üblicherweise nicht in einer Datei in dem Quellcode-Verzeichnis abgelegt werden. Bei komplexeren Konfigurationen kann man zur Not natürlich immer noch auf eine Konfigurationsdatei ausweichen. Trotzdem empfiehlt sich dann, den Pfad zur Konfigurationsdatei über eine Umgebungsvariable konfigurierbar zu machen. Auf dem Weg lassen sich nämlich auf einer Maschine mehrere Instanzen einer Anwendung starten, ohne dass diese sich gegenseitig in die Quere kommen. Die elfte Regel besagt, dass Logging stets auf den Standardausgabe- beziehungsweise Standardfehlerstrom der Anwendung erfolgen sollte. Das wirkt zunächst kontraintuitiv. Trotzdem ist es sinnvoll, da spätestens beim Betrieb der Anwendung in einem Container Log-Dateien flüchtig sind: Wird der Container zerstört, wird auch die Log-Datei verworfen. Daher ist es ratsam und in einem verteilten System auch einfacher und übersichtlicher, die verschiedenen Ausgabeströme der einzelnen Dienste zentral zu sammeln und auszuwerten. Diese Aufgabe lässt sich gut außerhalb der Anwendung durchführen, was die Entwicklung der Anwendung an sich wiederum einfacher macht. <figure class=„video akwa-inline-video video–fullwidth“><figcaption class=„bildunterschrift akwa-caption“>Konfiguration und Logging</figcaption></figure> Die fünfte Regel besagt, dass man das Bauen und das Ausführen einer Anwendung in zwei verschiedene Phasen zerlegen soll. Eine Anwendung wird zunächst erstellt und beispielsweise in ein Installationspaket oder ein Docker-Image verpackt. Anschließend wird sie ausgeführt. Es sollte für die Ausführung nicht mehr erforderlich sein, zunächst noch Code zu kompilieren oder Ähnliches – denn das verschlechtert nicht nur die Startzeit, sondern führt auch zu fragileren Anwendungen. Die zehnte Regel schließlich besagt, dass die Entwicklungs- und die Ausführungsumgebung möglichst gleich sein sollen. Damit ist beabsichtigt, dass man Fehler möglichst früh (also bereits während der Entwicklung) aufspüren kann, und nicht in der Staging- oder der Produktivumgebung die ein oder andere böse Überraschung erlebt, weil sich die Umgebung dort gravierend von der lokalen unterscheidet. Das Gleiche gilt natürlich nicht nur für die lokale Entwicklungs- und die Produktivumgebung, sondern auch für alle anderen Umgebungen, beispielsweise Test und QA. Auch dort sollte die Infrastruktur der finalen Umgebung möglichst gleichen. Das bedeutet natürlich nicht, dass alle Umgebungen auf die Datenbank aus dem Produktivsystem zugreifen sollten, aber es sollte eben lokal eine Datenbank zur Verfügung stehen, die der aus der produktiven Umgebung möglichst gleicht. <figure class=„video akwa-inline-video video–fullwidth“><figcaption class=„bildunterschrift akwa-caption“>Entwickeln, bauen und

betreiben</figcaption></figure> Wer sich an die 12 Regeln hält, macht im Hinblick auf eine tragfähige verteilte Architektur schon vieles richtig. Wie eingangs erwähnt, sind diese Regeln nicht die einzigen, die man bei komplexen Anwendungen im Sinn haben sollte, aber dennoch bilden sie eine gute Grundlage. Gerade, wer noch nicht über allzu viel Erfahrung in der Struktur und Architektur von Anwendungen verfügt, aber trotzdem eine verteilte Anwendung entwickeln will oder muss, der ist gut beraten, diese Regeln zu beachten, da sie sich in der Praxis bewährt haben. </html>