Sonderdruck aus JavaSPEKTRUM

Fluch oder Segen

Ein Erfahrungsbericht mit Microservices in Docker-Containern

Andreas Arnold, Martin Nitsch

In der Theorie klingt die Umsetzung von Microservices mit Docker immer ganz einfach, aber in der Praxis stößt man schnell auf zahlreiche Herausforderungen. Wie komme ich noch mal an meine Datenbank? Warum können die Services nicht miteinander kommunizieren? Wie lassen sich Docker-Container wiederverwenden? Bei der Implementierung einer Microservices-Architektur mit Docker kommt es zudem darauf an, sich schon frühzeitig über fachliche und technische Schnitte zu einigen – und deren Konsequenzen in den Blick zu nehmen. Der Artikel beleuchtet auf Basis vergangener Projekterfahrungen, welche Herausforderungen wir anfangs zu meistern hatten, welche technischen Schnitte wir vorgenommen haben, wie wir fachliche Module identifiziert haben und welche Vor- und Nachteile sich für uns aus dieser Architektur ergeben haben. Ein wichtiges Ziel für uns war auch die vollständige Automatisierung des Build- und Deployment-Prozesses mithilfe von Docker-Containern.

Die Philosophie der Microservices ist es, Verantwortlichkeiten und Funktionalitäten von Applikationen zu möglichst kleinen und fachlich sinnvoll unterschiedenen Einheiten zusammenzufassen und in unabhängige Services zu trennen. Diese Services sind nach dem Ideal der funktionalen Autonomie gestaltet und erfüllen häufig, aber nicht zwingend das Ideal der Autarkie im Sinne eines Self-contained Systems (SCS), das für Hochverfügbarkeitssysteme relevant ist.

Die Anforderung der Hochverfügbarkeit war zwar in unserem Projekt nicht gegeben. Für eine Microservices-Architektur fanden sich trotzdem gute fachliche und technische Argumente, die wir in diesem Artikel darstellen werden. Oft genannte Risiken und Hürden konnten wir durch gezielte Überlegungen zu Infrastruktur und Design überwinden.

Zunächst beschreiben wir die Build-Infrastruktur des Projekts, die wir zur Entwicklung der Microservices neu aufgebaut haben. Anschließend gehen wir auf die technischen und fachlichen Schnitte ein, die wir bei unserer Microservices-Architektur vorgenommen haben. Danach gehen wir auf die Vorteile von Microservices ein, die die Zusammenarbeit von verteilten Teams in einem Projekt erleichtern. Zum Schluss ziehen wir ein Fazit und fassen noch einmal die Gründe zusammen, die aus unserer Sicht für die Microservices-Architektur sprechen.

Build-Infrastruktur für Microservices

In einem agilen Projektumfeld mit Scrum-Teams wollen wir möglichst schnell ein *valuable integrated done increment* liefern können. Doch scheinen Mehraufwand und steigende Komplexität, die mit der Entwicklung von Microservice-Systemen und deren Deploy-

Andreas Arnold ist seit über 10 Jahren Softwareentwickler bei der andrena objects ag. Seine persönlichen Interessen liegen neben Softwarequalität auch in der Migration von Projekten hin zu Cloud-Anwendungen.
E-Mail: andreas.arnold@andrena.de

Martin Nitsch ist Junior-Software-Engineer bei andrena objects ag. Er interessiert sich für Microservices, Docker und CI.
E-Mail: martin.nitsch@andrena.de

ment verbunden sind, kurzen Auslieferungszyklen auf den ersten Blick eher zu widersprechen. Der Widerspruch lässt sich jedoch aus unserer Sicht auflösen und die erkannten Risiken lassen sich effektiv begrenzen.

Wer sich entscheidet, den Weg der Microservice-Entwicklung zu gehen, sollte ein klares Infrastrukturkonzept an der Hand haben – von der Einrichtung bis zum Betrieb –, um den Risiken gerecht zu werden. Gelingt die Umsetzung der Infrastruktur, winken Vorteile des Microservice-Konzepts, die gerade auch für die Dynamik agiler Projekte von Nutzen sind. Mitunter kann man Beschleunigungseffekte erleben.

Unsere Vorgehensweise orientierte sich an bereits bekannten Prinzipien der agilen Softwareentwicklung. Denn Konzepte wie Infrastructure as Code und Continuous Deployment and Integration (CDI) lassen sich auch für die Entwicklung von Microservices nutzen, um die vollständige Automatisierung des Build- und Deployment-Prozesses umzusetzen. Als Schlüsseltechnologie kommt hier Docker zum Einsatz.

Unser erstes Ziel war es, den Build-Server aufzusetzen, damit wir die CI sicherstellen können. Dazu nutzen wir Vagrant, um die virtuelle Maschine (VM) aufzusetzen. Einen Jenkins gibt es schon als Docker-Container, die CI hatten wir mittels Jenkinsfile als Build-Pipeline schnell umgesetzt.

Damit das Testsystem und später auch die Demo (für den Scrum-Review) und Produktion in jeweils eigenen virtuellen Maschinen laufen können, brauchten wir noch ein Tool, um die fertigen Docker-Container versioniert abzulegen. Hierfür nutzen wir Nexus, den es glücklicherweise auch als fertigen Docker-Container gibt. Entsprechend war die Jenkins-Pipeline schnell um ein Docker Push erweitert und besteht nun aus den Schritten (Stages) *Git Checkout -> Build -> Test -> Build Docker Image -> Push Docker Image*.

Als Nächstes wollten wir nicht nur Continuous Integration, sondern auch Continuous Deployment. Das vorhin gebaute Docker-Image soll also automatisiert auf dem Testsystem gestartet werden. Hierzu nutzen wir eine getrennte Build-Pipeline. Dieser Build kann nun entweder einen einzelnen Microservice oder aber das gesamte System auf einer unserer Stages ausbringen, also Test, Demo oder Produktion. Damit nicht für jede Stage ein eigener Container gebaut werden muss, übergeben wir dem Container eine Umgebungsvariable, anhand derer die passende Konfigurationsdatei ausgewählt wird.

Die virtuellen Maschinen, auf denen nun das System läuft, werden ebenfalls über Vagrant aufgesetzt. In einer Docker Compose-Datei ist konfiguriert, welche Services laufen. Der Jenkins-Job startet die neuen Docker-Container dann automatisch mittels SSH. Somit ist auch ein Continuous Deployment gewährleistet.

Dieser Aufbau der Infrastruktur hat wichtige Konsequenzen, die wir an dieser Stelle kurz andeuten wollen:

- Die Infrastruktur lässt sich leicht vom Deployment eines auf viele Microservices ausbauen und skalieren. Dies stellt die Erweiterbarkeit sicher.
- Versionierte Container sind klar definierte Laufzeitumgebungen und derselbe Microservice-Container kann vom Archiv auf Test-, Demo- und Produktivsystem ausgebracht werden. Dies stellt sicher, dass die Anwendung auch automatisch auf allen Systemen lauffähig ist.
- Bei Bedarf können ältere Versionen eines Microservice per Knopfdruck re-deployt werden. Somit hat man für Notfälle die Möglichkeit eines Rollbacks.

Technische Schnitte

Da wir uns entschieden hatten, eine Single-Page Application mit Angular 6 zu bauen, ergaben sich die ersten technischen Schnitte ganz automatisch durch die unterschiedlichen Technologien:

- Frontend: Hier nutzen wir ein mit der Angular-CLI erstelltes TypeScript-Projekt.
- Backend: Hier nutzen wir eine Spring-Boot-Applikation, die REST-Services bereitstellt.
- Datenbank: Wir haben uns für eine PostgreSQL entschieden.

Jede dieser Applikationen wird in einen eigenen Docker-Container gebaut (die PostgreSQL gibt es zum Glück schon fertig, da muss man nur ein wenig konfigurieren). Die Kommunikation zwischen Frontend und REST-Service bereitete dabei keine großen Schwierigkeiten, da die URL der REST-Schnittstellen von außen sichtbar konfiguriert war. Dies ist nötig, da der Zugriff von dem Browser aus auf einem lokalen Rechner erfolgt. Damit nicht zu viele verschiedene Ports nach außen geöffnet werden müssen, nutzen wir nginx. Abhängig von der URL wird dann die Anfrage an den passenden Microservice weitergeleitet.

Als etwas schwieriger stellte sich die Konfiguration der Datenbanken heraus. Hier wollten wir einen Zugriff von außerhalb der VM verhindern, nur der REST-Service sollte Zugriff auf die Datenbank haben. Doch auch hierfür bietet Docker eine Lösung. Statt den internen Port nach außen zu mappen, nutzen wir ein konfiguriertes Netzwerk, sodass nur der zugehörige REST-Service im selben Netzwerk liegt und Zugriff auf die Datenbank hat.

Die – aufgrund der unterschiedlichen Technologien – notwendigen technischen Schnitte bringen in der Praxis auch noch zusätzliche Vorteile. So ist es zum Beispiel problemlos möglich, das Backend zu aktualisieren, ohne dass der Anwender bei der Arbeit etwas davon mitbekommt. Natürlich gibt es eine Verzögerung, da Spring Boot einige Sekunden beim Start benötigt, aber der aktuelle Zustand innerhalb des Frontends bleibt vollständig erhalten. Und da REST stateless ist, kann sogar beim Ausfüllen eines mehrstufigen Formulars zwischendrin das Backend ausgetauscht werden.

Um die Anwendung später auf verschiedenen Stages ausbringen zu können, ist es erforderlich, die fertigen Docker-Container konfigurierbar zu bauen. Wir haben in der Docker Compose-Datei einen Umgebungsparameter an den Container übergeben: `STAGE=test` beziehungsweise `STAGE=prod`. Im Docker-Container wurde dann bei Spring Boot abhängig von der gewählten Stage die passende application.properties-Datei geladen. Im Frontend wurde ein ähnliches Prinzip genutzt: Es wird eine config.json-Datei geladen, die abhängig von der gewählten Stage ausgetauscht wird.

Fachliche Schnitte

Oftmals gibt es nicht nur technische Gründe, ein System zu scheiden. Wenn es voneinander unabhängige Bereiche gibt, bei denen gar keine oder nur wenige Daten sich überlappen, ist es auch sinnvoll, die Anwendung fachlich zu trennen. Ein Beispiel für eine oft sinnvolle fachliche Trennung wäre die Benutzerverwaltung. In vielen Anwendungen will man den Anwendern verschiedene Rechte zuweisen. Der Rest der Anwendung fragt nur nach, ob der angemeldete Benutzer auch berechtigt ist, eine Aktion durchzuführen. Die Überlappung der verschiedenen Teile ist also gering. Um solche Anwendungsteile zu identifizieren, eignen sich die Prinzipien des Domain-Driven Designs.

Die fachlichen Schnitte ziehen sich durch alle technischen Schichten, die wir im vorhergehenden Kapitel beschrieben hatten. Also hat die Benutzerverwaltung ein eigenes Frontend, einen eigenen REST-Service und eine eigene Datenbank.

Diese Trennung erhöht den Konfigurationsaufwand für die Docker-Container nur minimal, dafür ergeben sich einige Vorteile. Die Code-Basis für ein Teilsystem ist erheblich kleiner und damit wesentlich überschaubarer. Dies ermöglicht einen schnelleren Einstieg und auch eine verteilte Entwicklung. Wenn nötig kann sogar jeder Microservice von einem eigenen Team an einem anderen Standort entwickelt werden. Wir hatten einen regen Wechsel der Teammitglieder an mehreren Standorten. Diese Konstellation war nur durch diese fachliche Trennung der Teilsysteme effizient möglich.

Die Trennung der Datenbanken in mehrere kleinere bringt ebenfalls einen großen Vorteil. Man kann sich für jedes Teilsystem überlegen, welche Datenbank zu dem Problem am besten passt. So kann es durchaus an einer Stelle sinnvoll sein, auf eine spezielle NoSQL-Datenbank auszuweichen, weil man zum Beispiel viele Dokumente verwalten möchte. Die Benutzerverwaltung hingegen sollte eher in einer SQL-Datenbank liegen, da hier die Datenkonsistenz sehr wichtig ist und das Schema sich eher selten ändern wird.

Wichtig ist allerdings, dass zwischen den Teilsystemen nur relativ wenig Interaktion stattfindet. Stark verzahnte Systeme sind nicht nur schwer zu trennen, man bekommt dadurch auch deutlich mehr Probleme, als man lösen würde. Also sollte man sich vorher genau überlegen, welche Domänengrenzen es in der Software geben wird. Dann kann man gleich die erwarteten Interaktionen zwischen den Domänen aufmalen. Sind es zu viele, muss man das System anders schneiden. Vor allem zyklische Verbindungen sollte man auf jeden Fall unterlassen.

Verteilte Teams

Wir hatten bereits einige Vor- und Nachteile von Microservices-Architekturen angesprochen. Zumeist stehen bei solchen Betrachtungen die technologischen Aspekte im Vordergrund. Dabei wird übersehen, dass technologische Entscheidungen in der Softwareentwicklung die zwischenmenschliche Gestaltung und Organisation von Projekten und die Arbeit in Unternehmen nicht nur auf Anwenderebene, sondern ebenfalls auf Ebene der Entwicklung mitbestimmen.

Ein schlagender Vorteil, der aus unserer Sicht in diesem Zusammenhang mit Microservices-Architekturen verbunden ist, besteht darin, dass die Zusammenarbeit in verteilten Teams deutlich leichter gelingt und zu organisieren ist als bei verteilten Arbeiten an einem großen Monolithen. Die Transaktionskosten zwischen einzelnen Teams sind geringer, weil die Änderungen eines Teams den Code anderer Teams gar nicht betreffen. Notwendige Absprachen beschränken sich zum Beispiel auf die gemeinsame Einigung auf Pfade für REST-Schnittstellen. Auch die Zahl der Entwickler aus einem Team, die gleichzeitig zusammen an einem Code-Gefüge arbeiten, fällt insgesamt kleiner aus, und damit der Abstimmungsbedarf innerhalb eines Teams.

Dieser Punkt war gerade in unserem Projekt besonders wichtig, da wir dieses als Ausbildungsprojekt nutzen. Mitarbeiter von teilweise vier Standorten hatten gleichzeitig das Projekt weiterentwickelt. Hätten wir eine monolithische Architektur gewählt, wären Konflikte garantiert. Aber so konnten wir die Teams auf verschiedene Microservices aufteilen, sodass nach Absprache über die Schnittstellen keine Konflikte mehr auftreten. Falls Anpassungen an den Schnittstellen erforderlich waren, reichte das Daily Scrum aus, um die anderen Teams zu informieren.

Damit tritt auf Projektebene ein, was wir einen Beschleunigungseffekt nannten: Weniger Zeit muss in die Koordination und Organisation des Projekts investiert werden, sodass mehr Zeit für reine Entwicklungsaufgaben genutzt werden kann.

Fazit

Auch wenn der initiale Aufwand für ein Microservice-System mit Docker deutlich höher ist als ein klassischer Monolith, lohnt sich der Aufwand. Sobald man sich eingearbeitet hat und die initiale Konfiguration fertig ist, gibt es auch keine großen Hürden mehr, das laufende System zu erweitern.

Da man die gesamte Konfiguration von der virtuellen Maschine bis hin zu den Datenbanken vollständig automatisieren kann, ist auch ein Umzug von einem Rechner auf einen neuen unproblematisch. Auch die Zeit zwischen Anpassungen am Code bis hin zum laufenden Test-System ist deutlich geringer, man kommt deutlich unter 5 Minuten. Bei klassischen Monolithen würde dieses viele Stunden in Anspruch nehmen. Auch das Deployment von der aktuellen Testversion auf unserem Test-System auf das Produktiv-System läuft in weniger als einer Minute. Es sind keine manuellen Schritte notwendig, sodass man auch sicher sein kann, dass es keine Fehler beim Aktualisieren gibt.

Außerdem lässt sich die Arbeit bei verteilten oder großen Teams deutlich besser verteilen. Die Skalierbarkeit bezüglich der Entwicklung kann entsprechend auch eine Motivation sein, Microservices als Architekturmodell zu wählen. Es ist noch nicht mal erforderlich, dass immer dasselbe Team am selben Service entwickelt. Hier kann man gerne pro Sprint nach Bedarf wechseln, solange das fachliche Know-how ausreichend verteilt ist. Auch technisch dürfen sich die Services dann aber nicht zu sehr unterscheiden.

Falls viele Benutzer das System nutzen wollen oder sollen, kann die gute Skalierbarkeit der Microservices auch eine gute Motivation sein. Man kann schnell einen Load-Balancer einziehen und statt eines REST-Service mehrere starten. Dieses spielte bei unserem System keine Rolle, da man bei unter 50 Anwendern auch in ferner Zukunft keine Performanzprobleme erwarten kann. Doch in vielen anderen Anwendungen wird diese Überlegung eine Rolle spielen.

Ein weiterer großer Vorteil ist die Wiederverwendbarkeit der Services. Wir selbst nutzen zum Beispiel einen OAuth-Service für mehrere Anwendungen. Auch die Benutzerverwaltung kann man für mehrere Anwendungen wiederverwenden. Es gibt noch ausreichend weitere Beispiele von Microservices, bei denen sich eine Wiederverwendung lohnen würde (Mails schreiben, Logging, Monitoring usw.).

Aufgrund der vielen Vorteile einer Microservices-Architektur mit Docker können wir nur empfehlen, diesen Ansatz weiterzuverfolgen. Der Aufwand lohnt sich schon nach wenigen Wochen, und danach macht die Entwicklung deutlich mehr Spaß.