Richard Attermeyer
(richard.attermeyer@opitz-consulting.com)
arbeitet als Senior Solution Architect bei OPITZ CONSULTING. Er beschäftigt sich seit mehreren Jahren mit der Architektur und Implementierung von Anwendungen im agilen Umfeld. Er fokussiert sich dabei auf moderne Architekturansätze rund um Microservices, Cloud, DevOps und Continuous Delivery.

Marco Buss
(marco.buss@opitz-consulting.com)
ist Senior Consultant bei OPITZ CONSULTING und beschäftigt sich seit mehr als 10 Jahren mit der Java-Entwicklung im Enterprise-Umfeld. Seine aktuellen Themenschwerpunkte sind Continuous Delivery, Big Data und moderne Clients.

Ab in den Container! Continuous Delivery in der Praxis

Viele Unternehmen beginnen derzeit, sich mit Continuous Delivery zu beschäftigen. Häufig sind es einzelne Projekte, die solche Initiativen vorantreiben. Wird der Teilnehmerkreis größer, weil sich Abteilungen oder mehrere Teams dem Thema widmen wollen, braucht es ein umfassenderes Konzept: An welche Aspekte muss man denken? Wie baue ich die Umgebungen auf? Und wer übernimmt die Umsetzung?

Anforderungen an Continuous-Delivery-Vorhaben

Betrachten wir zunächst die Anforderungen, die Unternehmen an ein Continuous-Delivery-Projekt stellen: Typische Anforderungen, denen wir in den vergangenen Jahren immer wieder begegnet sind, lauten:

- Die verfügbaren Ressourcen für Builds sollen möglichst gut ausgenutzt werden.
- Die Bereitstellung von Staging-Umgebungen (dev, test, capacity, uat, e2e, …) soll automatisch erfolgen.
- Schulungsumgebungen müssen per Knopfdruck bereitgestellt werden können.
- Einheitliche Übergabe von Build-Artefakten an die Produktion, deren Deployment-Prozess bereits getestet wurde.
- Neue Projekte sollen automatisch auf den beteiligten Systemen eingerichtet werden (VCS, Wiki, CD Tool, …).
- Ein Fehler soll möglichst mit dem verursachenden Commit in Verbindung stehen.

In vielen Firmen existiert immer noch eine Sollbruchstelle zwischen der Continuous-Delivery-Umgebung und der Produktion. Das bedeutet, die Verantwortlichkeiten sind nicht einheitlich geregelt. Erst wenige Firmen wagen den Schritt, der Entwicklungsabteilung die Autonomie und das Vertrauen zu gewähren, per Knopfdruck in die Produktiv-Umgebung zu „deployen“.

Insbesondere der dynamische Aspekt und die Vereinheitlichung von Übergaben, implizieren die Nutzung von Container-Technologien. Container haben mit dem Siegeszug von Docker Einzug in die Bereitstellung und den Betrieb von Applikationen genommen. Doch welche Aspekte sind beim Aufbau einer solchen Umgebung zu berücksichtigen?

Vom Code zur ausführbaren Anwendung

Schauen wir uns eine typische Applikation an, bestehend aus einem Angular Frontend, einem Spring Boot Backend und einer MySQL-Datenbank. Das Deployment besteht also aus mindestens diesen drei Containern.

Wie aber wird aus einer Zeile Code ein Container? Im Folgenden betrachten wir das Vorgehen aus der Sicht eines neuen Entwicklers.

Schritt 1: Spezifikationen

Im Team-Wiki sind die Richtlinien für REST-Schnittstellen sowie Details zu den Userstorys dokumentiert. Bevor mit der Entwicklung begonnen werden kann, schaut sich der neue Entwickler erst einmal in den Dokumenten um.

Schritt 2: Versionskontrolle und Entwicklungsbox

Zunächst clont der Entwickler das Projekt vom zentralen Gitlab-Server. Im Projekt befindet sich ein Vagrantfile für das Aufsetzen der Datenbank. Dadurch ist gewährleistet, dass die Datenbank identisch zum Continuous-Delivery-Prozess ist.

Im Projekt wird Trunk-basierte Entwicklung betrieben. Für jede Story wird ein neuer Branch angelegt. Bei jedem automatischen Build wird der Trunk in den entsprechenden Branch gemerged. Dadurch werden Konflikte sehr frühzeitig sichtbar. Das verhindert erhöhten Merge-Aufwand am Ende einer Story. Dieses Vorgehen wird auch als „Pre-tested Commit Workflow“ bezeichnet.

Nachdem der Entwickler also den Branch für die zu bearbeitende Story ausgecheckt hat, kann er mit der Programmierung beginnen.

Schritt 3: Binär-Repository und Proxy
Um zu verhindern, dass Abhängigkeiten einfach aus dem Internet verschwinden [TRe], betreiben wir einen Proxy-Server. Risiken, die für private Projekte akzeptabel sind, müssen wir im Unternehmen absichern.

In unserem Beispiel hat sich das Team dazu entschieden, Sonatype Nexus zu verwenden, das als Proxy für Maven und NPM Repositorys fungieren kann. Eine Alternative wäre Artifactory von JFrog. Der Proxy stellt sicher, dass Versionen, die einmal im Internet angefordert wurden, danach im eigenen Netzwerk und unter eigener Kontrolle liegen. Wenn wir Librarys entwickeln, die von anderen genutzt werden sollen, dann müssen wir diese über ein entsprechendes Binär-Repository bereitstellen. Dies leistet Nexus ebenfalls.

Schritt 4: Continuous Delivery Server
Ist der Entwickler mit seiner lokalen Entwicklung zufrieden, kann er seine Änderungen zum Gitlab-Server pushen. Jeder Push löst dabei direkt einen Build-Prozess auf dem Continuous-Delivery-Server aus. So kann jeder Build genau einem Push zugeordnet werden. Enthält der Build mehrere Pushs verschiedener Entwickler, besteht die Gefahr, dass sich niemand verantwortlich fühlt, falls es zu Problemen während des Builds kommt.

Sobald der CI-Server den Trunk in den Branch „mergen" konnte, startet die eigentliche Build-Pipeline. Wie in vielen anderen Projekten, wird der CI-Server auch hier in einem Master-Slave-Modus betrieben, bei dem die Slaves Docker Container sind. Diese Container laufen auf einem Docker Cluster.

Durch die Verwendung von Containern kann jedem Build schnell eine neue, saubere Build-Umgebung bereitgestellt werden, die nicht durch vorherige Builds beeinflusst ist. Ein weiterer Vorteil ist, dass sehr dynamisch auf Last reagiert werden kann. Sind viele Builds in der Warteschlange, werden mehr Slaves erzeugt. Werden diese nicht mehr benötigt, können sie genauso schnell wieder abgebaut werden und verbrauchen dadurch nicht unnötige Ressourcen.

Moderne CI-Systeme wie Concourse setzen von vornherein auf einen Container-basierten Workflow.

Das Ergebnis eines Builds ist ein Docker-Image, das in einer privat gehosteten Docker Registry [DoR] abgelegt wird. Images wie auch die einzelnen Pipelines sind durch eine Tagversion gekennzeichnet, die aus einer Kombination von (gekürztem) Git Hash und Build-Nummer besteht, zum Beispiel „1035-ab12c453".

Schritt 5: Deployment der Anwendung (Docker Compose)
Von nun an arbeitet die Pipeline nur noch mit Docker-Images. Es wird immer ein kompletter Stack beschrieben, deployt durch einen Docker Compose File.

Hier greifen wir den bereits erwähnten dynamischen Aspekt auf. Für jede Umgebung kann der Stack anders aussehen. In einigen Umgebungen wird die Datenbank mit deployt, in anderen nicht. In den Testumgebungen werden gegebenenfalls spezielle Container verwendet, die das Rücksetzen einer Anwendung erlauben. Der Stack für die Produktivumgebung wird diese Funktionalität nicht benötigen.

Durch die Verwendung von Containern lassen sich die benötigten Umgebungen leichter bereitstellen. Es ist nicht mehr nötig, aufwendig, im schlimmsten Fall sogar manuell, eine oder mehrere VMs zu provisionieren, sondern der komplette Stack kann mit einem Kommando erstellt und auch wieder gelöscht werden.

Um ein Cluster von Docker Containern zu betreiben, gibt es verschiedene Lösungen wie Docker Swarm, Kubernetes, Nomad oder Mesos. Wir haben uns entschieden, bei der mitgelieferten Lösung zu bleiben und Docker Swarm einzusetzen.

Das Deployment eines Application Stacks auf einem Cluster bedeutet, dass die Komponenten beim ersten Deployment beispielsweise auf den Knoten 1, 3, 5 und 7 laufen, bei einem erneuten Deployment dann beispielsweise auf 2, 5, 7 und 9. Eine Anwendung ist daher aus Sicht des Clusters nicht direkt mit einer IP und einem festen Port verbunden. Die Deployment Pipeline sorgt dafür, dass eine vorhandene Umgebung zunächst entfernt und danach in der neuen Version gestartet wird.

Schritt 6: Service Discovery
Damit unser auf Protractor basierter E2E-Test den deployten Service findet, benötigen wir eine Service-Discovery-Funktionalität. In unserem Fall wird diese Funktionalität über Hashicorp Consul und Registrator realisiert. Die einzelnen Services sind dann für Skripte über die DNS-Schnittstelle von Consul einfach abzufragen. Durch einen Reverse Proxy werden mithilfe der Service-Discovery-Funktion feste DNS-Namen und Ports für den Zugriff bereitgestellt.

Product Owner und andere Stakeholder nutzen die deployten Umgebungen, um bereits während eines Sprints einen Blick auf die Anwendung zu werfen und Feedback zu geben. Auch hier zeigt sich ein großer Vorteil der Container und Stacks.

Das bisher beschriebene Vorgehen macht es möglich, für jeden Product Owner eine eigene Testumgebung bereitzustellen. Im Idealfall wird diese Umgebung nach den Tests sofort abgebaut und die Ressourcen werden wieder freigegeben.

Schritt 7: Manuelle Tests der Anwendung
Das Testing umfasst zum einen automatisierte Tests und zum anderen Tests durch

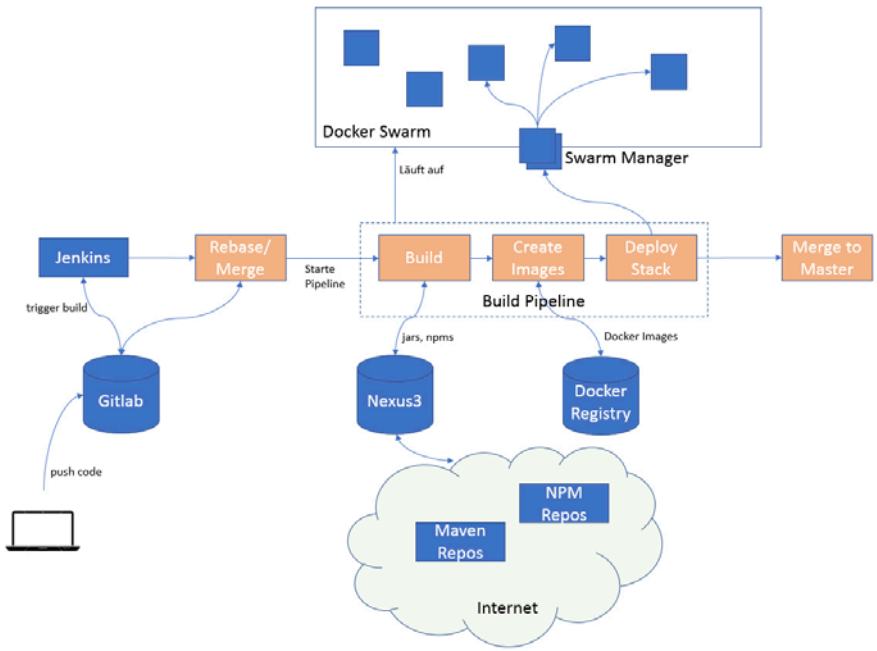

Abb. 1: CD-Prozess

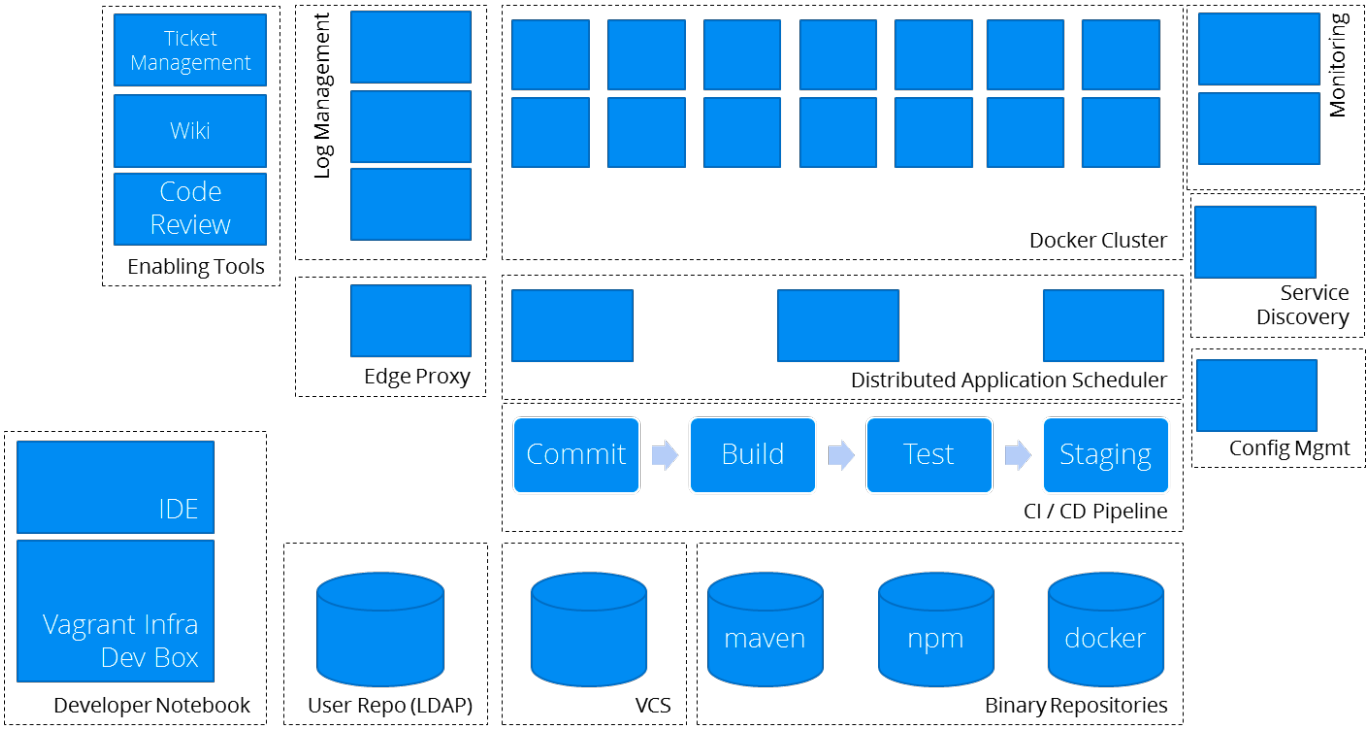

Abb. 2: Blueprint einer Continuous-Delivery-Umgebung

Business-Experten. Die Business-Experten halten Fehler in einem Ticketsystem fest, in unserem Fall in Atlassian Jira. Das Ticketsystem wird im Projekt also nicht nur als agiles Projektmanagement-Werkzeug genutzt sondern auch als Bug-Reporting-Werkzeug.

Schritt 8: Analyse eines Fehlers anhand von Logs und Metriken

Tritt während eines manuellen Tests der Anwendung ein Fehler auf, so wird eine Request-ID ausgegeben. Über diese ID kann eine Log-Management-Lösung alle zugehörigen Events ausmachen und analysieren.

Das dynamische Deployment macht ein Log Management notwendig. Die Informationen liegen hier nicht in einem zentralen Logfile und somit wäre die Analyse eines Problems sehr kompliziert. Über das Log Management, sowie über die Korrelation von Events über eine ID, lässt sich der zeitliche Verlauf einer Anfrage untersuchen, deren Log-Ausgaben sich über mehrere Dateien auf mehreren Knoten verteilen.

Neben Log-Nachrichten sind noch weitere Metriken einer Anwendung interessant. Dafür benötigen wir eine Monitoring-Software.

Klassische Monitoring-Lösungen wie Check_mk oder Icinga sind Host-basiert und passen daher nicht zu einem Cloud- oder Cluster-Ansatz mit dynamischen Umgebungen. In unserem Beispiel nutzen wir Prometheus als Monitoring-Werkzeug und Grafana zur Visualisierung. Durch geschickte Labels an den Docker-Containern können wir eine Applikation, ihre Komponenten und die einzelnen Umgebungen wieder zusammenfügen. So erhalten wir beispielsweise den Zeitverlauf der CPU oder den Speicherverbrauch der Anwendung, egal auf welchen Knoten sie gelaufen sind, und können damit Tendenzen feststellen. Mit klassischen Monitoring-Lösungen ist dies nur sehr schwer möglich.

Ein Blueprint für Continuous-Delivery-Prozesse

Abbildung 1 zeigt den Prozess und die Komponenten, die uns bei dieser kleinen Reise vom Entwickler-PC bis zur deployten Anwendung begegnet sind.

In unserem Beispiel haben wir einen Mix aus Komponenten verwendet, die aufgrund der gewachsenen Strukturen teilweise schon vorhanden waren. Unsere konkrete Instanziierung des Blueprints entspricht damit der Darstellung in Abbildung 3.

Der Vorteil dieses Blueprints ist, dass man mit ihm verschiedene Stacks durchspielen kann. Wer möglichst einfach anfangen möchte, kann, wo es geht, Gitlab nutzen und wird damit schon ziemlich weit kommen.

Vorteile fürs Business

Wie überzeugt man das Management von den Vorteilen eines Continuous-Delivery-Projekts? Welche Profits kann ein Unternehmen von einer solchen Umgebung erwarten?

Um eins vorwegzunehmen: Einen Projektmanager wird man schwerlich dazu bekommen, ein einzelnes Continuous-Delivery-Projekt aufzusetzen. Eine solche Umgebung ist tatsächlich erst sinnvoll, wenn sie in mehreren Entwicklungsteams zur Anwendung kommt. Daher fällt die Entscheidung eine Ebene höher. Eine Umgebung lohnt sich erfahrungsgemäß ab 20 Entwicklern und Testern, die aktiv auf ihr arbeiten.

Dann kann eine solche Umgebung ihre Vorteile aber auch voll zur Geltung bringen:

- Da es keine dedizierten VMs gibt, werden die zur Verfügung stehenden Ressourcen besser ausgenutzt und eine höhere Packungsdichte erzielt.
- Es ist einfach möglich, weitere Umgebungen hinzuzufügen: Wurde früher eine Testumgebung vergessen, war meist erst ein langwieriger Prozess notwendig, bis die Umgebung zur Verfügung stand. Hier nutzen die Container einfach freie Ressourcen.
- Eine Erweiterung der Umgebung kommt direkt allen Projekten zugute. Werden dedizierte Ressourcen benötigt, so können einzelne Knoten im Cluster entsprechend getaggt werden.

Unter dem Strich bedeutet Continuous Delivery für das Unternehmen oder die Entwicklungsabteilung signifikante Einsparungen und eine erhöhte Flexibilität.

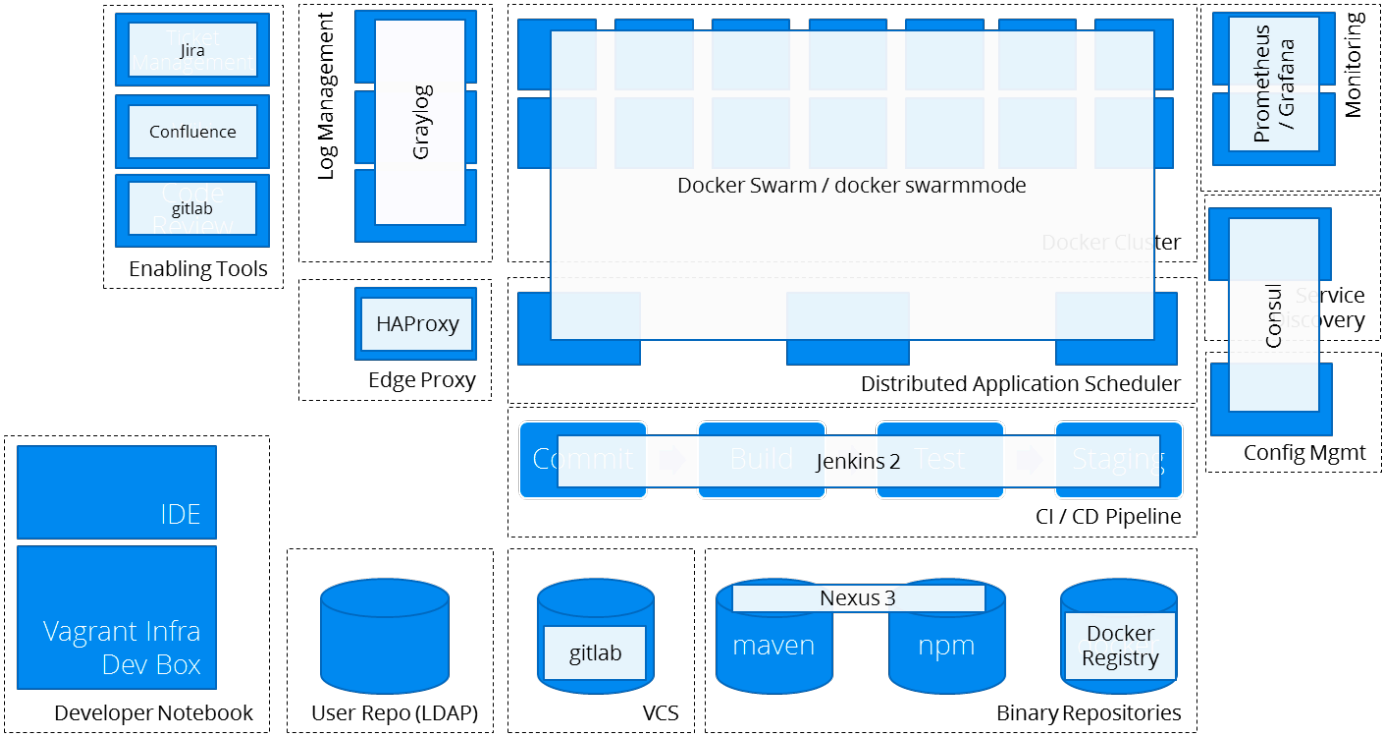

Abb. 3: Instanz einer Continuous-Delivery-Umgebung

Fazit

Durch den Einsatz von Containertechnologien haben sich im Bereich Continuous Delivery viele interessante neue Möglichkeiten ergeben. Vor allem die Vorteile der leichten Bereitstellung eines kompletten Application Stacks und die dynamischere Auslastung der vorhandenen Ressourcen sind als Vorteile zu nennen. ■

Referenzen

[TRe] How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript, https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

[DoR] https://docs.docker.com/registry/