JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

Continuous Delivery

Ausgabe 6.2017

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 06

Bot-basierte Kommunikation

DevOps mit ChatOps

Viele Softwareunternehmen versuchen zurzeit, DevOps zu implementieren und ihre Prozesse und Abläufe entsprechend zu optimieren. Im Zuge dieses Hypes halten verschiedene neue und wiederentdeckte Techniken Einzug in den Arbeitsalltag, eine davon ist ChatOps.

von Christian Kühn

Kommunikation ist der Austausch oder die Übertragung von Informationen [1]. Übertragung heißt, dass dabei Distanzen überwunden werden können. In diesen zwei Sätzen stecken mehrere essenzielle Informationen und Herausforderungen, mit denen wir im alltäglichen Geschäft von Softwareentwicklung und -betrieb konfrontiert werden. Grundlegend bedarf jeder Schritt der zwischenmenschlichen Kommunikation, wenn mehr als eine Person an der Lösung einer Aufgabe beteiligt ist. Nicht jeder Informationsaustausch kann zwischen zwei Personen im gleichen Raum stattfinden. Absprachen finden auch mit entfernten Kunden und zunehmend in verteilten Teams statt. Sowohl im kommerziellen als auch im Open-Source-Umfeld soll natürlich auch jede wichtige Information dokumentiert werden, ohne gleichzeitig einen übermäßigen Aufwand zu erzeugen.

Klassisch fand diese Kommunikation per Telefon und Fax statt, mittlerweile benutzen die meisten Firmen E-Mail oder Ticketsysteme. Für interdisziplinäre Teams, die teils räumlich verteilt zusammenarbeiten, ist das zu zeitaufwendig und kompliziert. Um Informationen schneller und transparenter zu verteilen, haben sich in vielen Projekten Chatsysteme etabliert, in denen offene Fragen für alle Teilnehmer sofort ersichtlich sind, sodass dem Autor direkt geantwortet oder geholfen werden kann. Zusätzlich ist klar, ob die Frage bereits beantwortet wurde oder die Antwort noch erweitert oder korrigiert werden sollte. Beispiele dafür sind: IRC (Internet Relay Chat, Text-only), Slack, HipChat, Mattermost, Rocket.Chat (Browserclient mit bunten Bildern) oder die Integrationen in anderen Kommunikationstools (Skype, Google Hangouts, Jabber/XMPP).

Im Chat geschriebene Kommunikation können auch Teammitglieder nachlesen, die erst zu einem späteren Zeitpunkt aktiv werden, da sie andere Arbeitszeiten haben oder sich in einer anderen Zeitzone aufhalten. Natürlich ist das auch bei E-Mail und Ticketsystemen gegeben, doch in einem Chatroom lassen sich unserer Erfahrung nach Informationen einfacher kontextbezogen austauschen. Zusätzlich gibt es hier oft schnelleres Feedback, weil Chatteilnehmer Fragen oder Anforderungen möglicherweise direkt beantworten können, wenn die adressierte Person gerade nicht verfügbar ist.

ChatOps: Was ist das denn nun?

Da sich viel Kommunikation im Chat abspielt, entstand die Idee, den Chat auch als Interface und Steuerungsmechanismus für andere Systeme zu nutzen. Der Vorteil besteht darin, dass man nicht erst im Chat über einen bevorstehenden Arbeitsschritt informieren muss, sondern diesen Schritt auch ausführt und das entsprechende Feedback automatisch im Chat persistiert wird, sodass andere Teilnehmer informiert sind und reagieren oder unterstützen können.

Die heute verbreitete Ausprägung von ChatOps wurde maßgeblich von GitHub vorangetrieben, dort wurde 2011 der interne Chatbot unter dem Namen Hubot neu implementiert, als Open-Source-Projekt bereitgestellt und aktiv für diese Arbeitsweise geworben [2]. Die Grundidee ist, einen Bot im Chat bereitzustellen, der auf bestimmte Keywords und Parameter lauscht und entsprechende Aktionen ausführt. Der Bot kann als eine geteilte Shell gesehen werden, in die alle Chatteilnehmer Befehle eingeben können (**Abb. 1**).

Es besteht die Möglichkeit, einem Bot einen Befehl zu geben, dessen Ausführungszeitpunkt, Syntax und Rückgabewert alle Chatteilnehmer sehen können. Der Chat wird damit zur Middleware zwischen verschiedenen Diensten, die alle zentral gesteuert werden können. Mit der Ausführung der Befehle oder Aktionen mittels Chat ist automatisch sichergestellt, dass alle Teammitglieder sehen können, was genau passiert ist.

Einfache Kommunikation, wenige Missverständnisse

Ein Beispiel ist ein Projekt, für das zu einem bestimmten Zeitpunkt händisch ein Release getriggert werden

soll. Je nach Aufbau einer vorhandenen CI/CD-Pipeline könnte man sich das folgendermaßen vorstellen:

- Eine neue Versionsnummer wird in eine Konfiguration wie Maven *pom.xml* eingetragen.
- Es wird ein Git-Tag mit dieser Versionsnummer angelegt und ins Code-Repository gepusht.
- Jenkins [3] wird benachrichtigt und beginnt mit dem jeweiligen Build.
- Das entstandene Artefakt wird in ein passendes Artefakt-Repository gespeichert. Wir benutzen hier abhängig vom Projekt Nexus [4] oder Artifactory [5].
- Das Artefakt wird von seinem Speicherort auf einen Server deployt.

Dieser komplette Ablauf ist typischerweise schon automatisiert, z. B. in einer Jenkins-Pipeline, und muss nur noch ausgelöst werden. Das kann auf verschiedene Arten passieren, je nach Ausbau der Automatisierung händisch, als Skript oder beispielsweise als Post-Receive Hook in Git. Der Aufruf dieser Releasepipeline mithilfe eines Chatbots könnte wie in **Abbildung 2** aussehen.

Die ausgeführte Funktion ist für das ganze Team und sonstige Chatteilnehmer leicht zu erkennen: Der Benutzer *chris* hat um *10:19 Uhr* eine Funktion *release* mit den Parametern *DemoProjekt* und *1.0* ausgeführt. Der Bot hat sofort angezeigt, dass die Eingabe erkannt wurde. Nach Beendigung des Jobs zwei Minuten später hat er eine positive Rückmeldung in den Chat geschrieben. Zusätzlich lässt sich diese Meldung natürlich auch in weitere Chatrooms schreiben.

Hätte der Benutzer die Pipeline händisch auf seinem Rechner gestartet, wäre es nötig gewesen, die anderen Teammitglieder von dieser Aktion in Kenntnis zu setzen. Er hätte also neben der eigentlichen Aktion in den Chat schreiben müssen, wann er diese Aufgabe erledigt hat und möglicherweise auch den jeweiligen Return-Wert oder die Fehlermeldung. So hat der Benutzer nicht nur seine eigene Zeit gespart, weil er nicht zwischen seiner lokalen Shell, dem Jenkins-GUI und dem Ticketsystem hin- und herschalten muss, vor allem sehen auch alle anderen Benutzer sofort die nötigen Informationen. Das vermeidet Missverständnisse, die entstehen können, wenn Arbeitsschritte doppelt ausgeführt werden, z. B. weil ein Chatteilnehmer die entsprechende E-Mail nicht erhalten oder das jeweilige Ticket falsch zugewiesen oder einsortiert war.

Diese Abstimmungsschwierigkeiten sind im Bereich der Softwareentwicklung weniger problematisch, weil man Funktionen gegen mehrfache Ausführung absichern kann. So sollte ein Entwickler bemerken, wenn seine Projektkonfiguration bereits eine neue Versionsnummer hat. Spätestens bei einem Merge-Konflikt gibt moderne Versionierungssoftware einen Hinweis. Im Operations-Bereich hingegen, wo vor allem auf Events reagiert werden muss, können mehrfache Ausführungen des gleichen Arbeitsablaufs zeitraubend sein. Man stelle sich folgendes Beispiel vor:

```
14:44 <@chris> demoBot: sag mal hallo!
14:44 <@demoBot> hallo chris, ich bin demoBot
```

Abb. 1: Der Bot sagt hallo

```
10:19 < chris> release: DemoProjekt 1.0
10:19 <@demoBot> releasing DemoProjekt 1.0 to staging servers
10:21 <@demoBot> ✓ build 1.0 success, deploy success
```

Abb. 2: Aufruf der Releasepipeline

- Kunde erstellt ein Ticket: „Server 15 ist zu langsam seit heute Vormittag“.
- Dienstleister prüft den Fall, kann weder auf dem Server noch im Monitoring ein Problem feststellen und eskaliert den Fall auf eine höhere Ebene.
- Technischer Support prüft die Virtualisierungsserver im gleichen Cluster, das Storage-System sowie andere VMs auf dem System und findet heraus, dass der Server zu wenig Arbeitsspeicher besitzt.
- Als kurzfristige Lösung wird der Server mit mehr Speicher ausgestattet.
- Während des folgenden Reboots findet der Support das Ticket, in dem ein Deployment von DemoProjekt 1.0 beschrieben ist.

Hier wurde ein erheblicher Aufwand generiert, da die Kommunikation zwischen verschiedenen Teams nicht funktioniert hat. Wahrscheinlich hätte eine kurze Mitteilung dem Support geholfen, die Probleme auf eine neue Version des Projekts zurückzuführen.

Neben der erweiterten Transparenz und dem Zeitgewinn, der durch die öffentlich sichtbare Ausführung eines Befehls entsteht, bietet ein Chatbot weitere Vorteile. Hubot dient als eine teaminterne Shell, in die jeder seine private Skriptsammlung einpflegen kann, um den anderen Mitgliedern die Nutzung zu ermöglichen. Es geht nicht nur darum, Skripte zum Kopieren bereitzustellen. Dazu würde auch ein zentrales Share ausreichen. Die Ausführung dieser Befehle schafft bei anderen Teammitgliedern das Bewusstsein, dass gewisse Befehle und Abläufe überhaupt existieren und mit welchen Parametern man sie effizient ausführt. Das ermöglicht geringere Einarbeitungszeiten neuer Mitarbeiter oder bestehender Mitarbeiter in neue Bereiche, weil man bestimmte Frameworks und Systeme nicht unbedingt komplett kennen muss. Auch ohne das genaue Wissen, unter welchem Hostnamen welcher Service installiert ist, kann man schon einfache Befehle darauf ausführen. Der Wissenstransfer entsteht hier quasi nebenher durchs Mitlesen. Damit lassen sich auch bestimmte Aufgaben an nicht technische Abteilungen und Mitarbeiter abgeben.

DevOps auch für Nicht-Techies

Wir ermöglichen beispielsweise den Mitarbeitern des Officemanagements, direkt mithilfe von Hubot Accounts für neue Mitarbeiter anzulegen. Dazu müssen in diversen Kategorien unseres Ticketsystems neue Tickets angelegt werden, z. B. zur Hardwarebeschaffung, aber auch zu organisatorischen Themen. Zusätzlich müssen

```
15:44 <@chris> erstell Benutzer namens Max Mustermann 01.05.2017
15:44 <@demoBot> Danke für die Info, ich leg dann mal die Accounts an
```

Abb. 3: Auch Accounts für neue Mitarbeiter lassen sich mit dem Chatbot anlegen

```
11:52 < demoBot> [Jenkins] build success DemoProject:#7246
```

Abb. 4: Jenkins meldet den erfolgreichen Build

```
10:58 < demoBot> [NAGIOS] demo-stage.synyx.de OutOfMemory
```

Abb. 5: Auch Warnungen kann der Bot direkt in den Chat schreiben

diverse Accounts angelegt werden, z. B. E-Mail-Adresse und LDAP-Benutzeraccount. Mithilfe der Keywords *erstelle Benutzer namens* wird die konfigurierte Aktion des Chatbots mit den Parametern *Max Mustermann* und *01.05.2017* ausgeführt (**Abb. 3**). Entsprechende Unteraufgaben werden während der Abarbeitung im Chat kommentiert, inklusive eventueller Zusatzinformationen wie Ticketnummern und Fehlermeldungen.

Ohne ChatOps musste für diese Aufgabe ein Ticket an die Systemadministration gestellt werden, die alle weiteren Schritte übernahm. In der aktuellen Variante werden nebenher automatisch alle Chatteilnehmer auf den neuen Mitarbeiter aufmerksam gemacht. Eventuelle Fehler sind direkt für den Systemadministrator ersichtlich. Bei einer fehlerhaften Eingabe des Benutzers kann auf die korrekte Eingabesyntax hingewiesen werden. Außerdem passiert die Durchführung sofort und nicht erst, wenn jemand aus einer bestimmten Abteilung Zeit hat oder auf das entsprechende Ticket aufmerksam wird.

Keiner starrt mehr aufs Dashboard

Ein weiteres Feature von Hubot ist der asynchrone Empfang von Benachrichtigungen sowie deren Weitergabe in einen Chatroom. Hubot kann bei entsprechender Implementierung HTTP-Endpunkte anbieten, über die man von außen Informationen in den Chat einfließen lassen kann. Diese Endpunkte könnten beispielsweise von Jenkins als Webhook angesprochen werden und nach jeder Jobausführung das Ergebnis in den passenden Chat propagieren (**Abb. 4**). Wir triggern in manchen Projekten Build-Jobs und komplette Pipelines über den Chat. Falls ein solcher Job eine sehr lange Laufzeit hat, empfiehlt es sich, die Rückmeldung des Build-Tools in diesen Mechanismus in den Chat zu schreiben und vom ursprünglichen Aufruf des Jobs zu entkoppeln.

So reicht es, wenn ein Entwickler nur den Chat beobachten und nicht ständig im Jenkins-Webinterface nachsehen muss, ob der Job noch läuft, ob er erfolgreich abgeschlossen wurde oder Fehler aufgetreten sind. Mit dieser Funktionalität lassen sich auch Alarme in den Chat integrieren, beispielsweise schickt ein Monitoringserver seine Warnmeldungen an einen Endpunkt des Bots (**Abb. 5**).

Sicherheit: Authorized Personnel only

Durch die Bereitstellung eines zentralen API für verschiedene Services spielt natürlich die Sicherheit eine große Rolle. Es sollte unbedingt eine Einschränkung auf Benutzerebene eingeplant werden. Je nachdem, welches Chatframework im Projekt zum Einsatz kommt, werden bereits Authentifizierungsmechanismen mitgeliefert. Die meisten modernen Chatsysteme bieten eine Log-in-Funktion, die einer Person eine eindeutige Identität zuweist. Im Beispiel von Rocket.Chat [6] lässt sich dieser Log-in über LDAP realisieren. Man kann also ein bestehendes Usermanagement weiterhin nutzen. Hubot kann in diesem Fall abhängig vom Nickname oder einer ID des Benutzers dessen Rollen vom LDAP-Server abfragen und bei jedem Befehlsaufruf prüfen, ob der Benutzer die nötigen Rechte besitzt. Im Fall von IRC kann man sich mit NickServ behelfen, einem Service, der die Eindeutigkeit von Benutzern über seinen Nickname sicherstellt. Auch diese Methode würde es erlauben, die Rollen dieses Benutzers gegen Daten aus LDAP zu prüfen. Die einfachste Form von Authentifizierung ist es, dem Bot eine private Nachricht mit einem Key zu schicken, um sich direkt anzumelden, und die Rollen in der Konfiguration des Bots zu definieren. Bei IRC-Bots wie Eggdrop [7] wird dieses Prinzip seit über zwanzig Jahren genutzt.

Eine weitgehende Integration von ChatOps erlaubt im Gegenzug eine gewisse Zentralisierung der Infrastruktursicherheit, da Benutzer nur noch Befehle über den Bot absenden, ohne selbst Zugang zu einem Zielsystem zu benötigen. Da die Bot-Syntax nur die echten Use Cases eines Projekts oder einer Firma abbildet, reduziert sich die Anzahl potenzieller Fehlerquellen. Über die Implementierung kann man für jeden Befehl eigene Grenzwerte festlegen, innerhalb derer die Aktion für bestimmte Nutzer über den Bot ausführbar ist.

Es empfiehlt sich, mehrere Instanzen des gleichen Bots zu benutzen, die über ihre Konfiguration nach Fachlichkeit oder Team getrennt sind. Die Zugriffsrechte des Bots und seine Zielsysteme werden über die lokale Konfiguration der jeweiligen Instanz gesteuert. Das stellt sicher, dass Funktionen und Keywords für alle Teams gleich aussehen, im Hintergrund aber die jeweiligen spezifischen Systeme ansprechen. Dieses Prinzip erlaubt es, eine gemeinsame Codebase zu implementieren, aus der sich alle Bot-Instanzen bedienen. Somit kann der Aufwand der Implementierung neuer Features auf alle Mitarbeiter verteilt werden. Anhand des Beispiels Jenkins sieht die Implementierung dann so aus, dass die Konfiguration einer Instanz festlegt, welcher Jenkins-Server angesprochen wird und welche Jobs auf diesem Server ausgeführt werden dürfen.

Als Vision wäre es denkbar, Zugriff auf APIs und Server z. B. per SSH komplett auf den Bot einzuschränken, um weitere Fehlerquellen und Angriffsvektoren einzuschränken. Man sollte beachten, dass mit der wachsenden Benutzung und Integration des Chatsystems ein Ausfall größere Auswirkungen hat. Daher sollten entsprechende Systeme redundant ausgelegt und entsprechend überwacht werden.

Logging: Wer? Wann? Was?

Um im Fall eines Fehlers, einer Falscheingabe oder eines anderen Problems ein schnelleres Troubleshooting zu

erlauben, empfiehlt es sich, ein Auditskript einzubinden. Man sollte alle Aufrufe des Chatbos, den zugehörigen Benutzer und den jeweiligen Output protokollieren. So lassen sich auch sicherheitskritische Aufrufe genauer nachprüfen oder kontrollieren. Zusätzlich erlaubt ein Auditing eine gezieltere Weiterentwicklung des Bots. Mit einer einfachen Analyse kann man herausfinden, welche Befehle oft benutzt werden und welche oft syntaktisch falsch aufgerufen werden, weil sie möglicherweise nicht eindeutig beschrieben sind. Da man mit Hubot jegliche HTTP-basierten APIs ansprechen kann, ist es auch mit wenig Aufwand möglich, Ticketsysteme einzubinden, um zur Dokumentation automatisiert Tickets bei bestimmten Aufrufen anzulegen und z. B. dem Aufrufer oder der passenden Abteilung zuzuweisen.

Wie komme ich selbst zu ChatOps?

Eine der bekanntesten Implementierungen eines Chatbots ist das GitHub-Projekt Hubot. Hubot ist in CoffeeScript geschrieben. CoffeeScript wird zu JavaScript kompiliert. Dadurch kann Hubot auch nativ mit JavaScript umgehen. Alternativen zu Hubot existieren in allen erdenklichen Sprachen. Im (Dev-)Ops-Bereich erfreuen sich Lita und Err größerer Beliebtheit. Sie sind in Ruby bzw. Python geschrieben und liegen damit näher an den Aufgaben und Anforderungen, die im Linux- und Automatisierungsumfeld bestehen. Hier werden Frameworks wie Chef oder Ansible eingesetzt, die vor allem in diesen Sprachen gepflegt werden. Letztendlich empfiehlt es sich natürlich, einen Bot in der Programmiersprache auszusuchen, in der im Unternehmen oder Projekt schon eine gewisse Grunderfahrung besteht. Dadurch kann man eine breite Plattform zum Mitentwickeln schaffen, ohne zusätzliche Lernaufwände zu generieren.

Für Hubot gibt es zum einfachen Testen ein Node.js-Paket auf Basis von Yeoman [8], einem Codegenerator für Node-Projekte. Das Paket lässt sich mithilfe von npm [9] installieren, einem Paketmanager für JavaScript. Es bietet einen leicht verständlichen Wizard, mit dem man seinen ersten Bot konfigurieren und starten kann:


```
npm install -g generator-hubot
yo hubot
```

Im Wizard werden die Standardeinstellungen festgelegt, wie der Nickname, den der Bot im Chat benutzen soll. Außerdem wird der Adapter konfiguriert, der das API für den Chat der Wahl anbindet. Für die gängigen Chatsysteme existieren bereits npm-Pakete, die hier automatisch geladen werden. Zusätzliche Features können bei Hubot unkompliziert als Erweiterungen per npm mit *npm install hubot-audit* installiert werden. Mit *hubot-audit* lassen sich die Aufrufe aller Hubot-Funktionen in einem Logging-Chatroom protokollieren. Alternativ zu npm kann man die genutzten Pakete in der Datei *package.json* im *hubot*-Folder eintragen. Die Inhalte dieser Datei werden als Dependency beim Start installiert. Selbstgeschriebene Skripte landen bei Hubot im *./scripts/*-Folder und werden beim Start geladen. Typischerweise sind diese Skripte in CoffeeScript oder JavaScript geschrieben. Ein Hello-World-Skript könnte folgendermaßen aussehen:

```
robot.respond /sag mal hallo!/, (res) ->
  issuer = res.message.user.name
  res.send "hallo #{issuer}, ich bin #{robot.name}"
```

Der Bot lauscht (*respond*) in allen Chatrooms darauf, dass er wie ein regulärer Chatteilnehmer mit seinem Namen angesprochen wird. Der Funktionsaufruf wird hier auf den regulären Ausdruck *sag mal hallo!* gematcht. Der Aufruf muss also genau diesen String enthalten, um die Funktion auszulösen.

Die konkrete Implementierung unseres Bots versionieren wir in einem zentralen Git Repository, auf dem alle Mitarbeiter neue Funktionen implementieren können. Die jeweiligen Instanzen werden per Push Deploy ausgerollt. Hierzu ist auf

der eigentlichen Hubot-VM ein Git Remote eingerichtet, das den Bot über einen Post Receive Hook neustartet, wenn neuer Code gepusht wird. Zur Absicherung einer Funktion kann und sollte eine Autorisierung implementiert werden. Man kann den Aufruf z. B. mithilfe der in Hubot mitgelieferten Middleware [10] absichern und die Funktion um einen Parameter erweitern:

```
robot.respond /sag mal hallo!/, auth: {group: 'sysadmin'}, (res) ->
```

In diesem Fall wird durch unsere Implementierung von *auth* geprüft, ob dem ausführenden User die LDAP-Rolle *sysadmin* zugewiesen ist. Diese Middleware empfängt alle Aufrufe des Bots und veranlasst dann die eigentliche Ausführung der jeweiligen Funktion. Dadurch ist es möglich, vor der Ausführung einer Funktion verschiedene Parameter auszuwerten, um festzustellen, ob die eigentliche Aktion ausgeführt werden darf. Neben der Identität oder Rolle eines Benutzers lassen sich auch andere Voraussetzungen prüfen, beispielsweise der Chatroom, in dem das Kommando ausgeführt wurde, oder wie oft das Kommando in einem Zeitraum schon gelaufen ist. Durch die freie Implementierung sind der Kreativität hier keine Grenzen gesetzt.

Zusätzlich lassen sich über die Middleware zwischen Aufruf und Ausführung einer Funktion noch andere Aktionen durchführen. Ein Beispiel dafür findet man in unserem Skript *hubot-audit*. In diesem Fall protokollieren wir für jeden Aufruf einer Hubot-Funktion die Syntax des Aufrufs, den Namen des Aufrufers und seinen Chatroom sowie zusätzlich die jeweilige Antwort des Chatbots, um bei Problemen schneller die Ursache zu finden.

Integration in die bestehende CI-/CD-Pipeline

Wir binden mittlerweile neue Funktionen vorrangig über Rundeck [11] ein. Die Anbindung von Hubot an Rundeck realisieren wir über das Rundeck-REST-API. Hubot übernimmt dabei nur noch die Orchestrierung von Jobs, die in Rundeck konfiguriert werden. Diese Jobs lassen sich unserer Erfahrung nach einfacher definieren und überwachen als Hubot-Skripte. Außerdem kann man die Jobs untereinander in Abhängigkeit setzen und automatisiert ausführen. Neben der manuellen Steuerung bringt Rundeck viele weitere Features wie automatisches Scheduling mit.

In der Softwareentwicklung ist vor allem auch die Anbindung an Jenkins interessant. Hubot kann wie bei Rundeck über das API Jenkins-Jobs und -Pipelines steuern. Diese Jobs sind in der Regel ohnehin schon als Teil der CI/CD Toolchain vorhanden und lassen sich in Jenkins komfortabel pflegen und überwachen. Hubot dient in Jenkins wie auch bei Rundeck nur als zusätzliche Steuerungsmöglichkeit.

Lessons learned: Was will man nicht machen?

Es gibt wie bei vielen Open-Source- und Communityprojekten eine Fülle an Paketen und Features, die man in seine eigene Implementierung einbinden kann. Während dieser Artikel verfasst wurde, wurden auf www.npmjs.org über 200 – 300 Hubot-Plug-ins angeboten, die alles Erdenkliche von komplexen Deployments bis hin zum Generieren von ASCII-Katzenbildern übernehmen. Letzteres zeigt die Gefahr, dass man sich schnell zu viele Abhängigkeiten ins Haus holt und die gewonnene Produktivität eher in Zeitverschwendung und Ablenkung umschlägt.

Wir haben in unserer Produktivumgebung versucht, auf ablenkende Bot-Funktionen zu verzichten, die nicht unbedingt das Tagesgeschäft unterstützen. Wer darauf nicht verzichten will, sollte es auf einen Off-Topic-Channel reduzieren. Durch den stärkeren Fokus auf den Chat während des Tagesgeschäfts ist die Gefahr größer, sich durch unnötige Information, externe Links, Katzenbilder und Ähnliches ablenken zu lassen. An dieser Stelle ist eine gewisse Selbstkontrolle der Mitarbeiter gefragt, damit die Vorteile einer regelmäßigen Chatbeobachtung nicht untergehen.

Wenn man ChatOps neu einführt, kann es auch einfacher sein, eine vorhandene Automatisierungs- oder Integrationslösung anzusprechen. Für den Einstieg kann man sich z. B. darauf beschränken, Jenkins, Rundeck oder andere Frameworks einzubinden, um bereits vorhandene und bewährte Jobs zu triggern und deren Ergebnis wieder im Chat zu propagieren. Als weiteren Vorteil haben wir festgestellt, dass eine größere Awareness für den Chat geschaffen wird, die Kollegen schneller ansprechbar sind, weil sie einfach mehr Zeit im Chat verbringen, in der sie vorher im Ticketsystem oder E-Mail-Client gestöbert haben. Zusätzlich entsteht hier auf eine spielerische Art ein Wissenstransfer, im Sinne von DevOps, wenn Entwickler die Ops-bezogenen Aktionen im Chat kennen lernen und auch selbst ausführen können.

Christian Kühn ist als Systementwickler bei der synyx GmbH & Co KG in Karlsruhe tätig und beschäftigt sich dort mit Softwareentwicklung und Systemadministration.

@CYxChris

Links & Literatur

[1] Definition Kommunikation: https://de.wikipedia.org/wiki/Kommunikation

[2] https://github.com/hubot

[3] Jenkins: https://jenkins.io

[4] Nexus: http://www.sonatype.org/nexus

[5] Artifactory: https://www.jfrog.com/artifactory/

[6] Rocket.Chat: https://rocket.chat/

[7] Eggdrop: https://www.eggheads.org/

[8] Yeoman: http://yeoman.io/

[9] npm: https://www.npmjs.com/

[10] Listener Middleware: https://github.com/github/hubot/blob/master/docs/scripting.md#listener-middleware

[11] Rundeck: https://rundeck.org