15 DevOps

Als ich mit dem Programmieren anfing, war meine Aufgabe klar: Software erstellen und zur Freigabe übergeben. Nach der Übergabe sollte ein geheimnisvoller Prozess die Software in die Hände der Kunden bringen. Zunächst ging es um den Versand von CDs, später war eine entfernte Abteilung namens »Betrieb« beteiligt, die von Vogelgesängen besessen zu sein schien (*awk! grep! perl!*). Wie auch immer, es ging mich nichts an.

Das blieb auch noch so, nachdem ich begonnen hatte, agile Methoden zu verwenden. Obwohl agile Teams funktionsübergreifend sein sollten, wurde der Betrieb von anderen Leuten abgewickelt – Personen, die ich nie traf und deren Namen ich selten kannte. Ich wusste, dass dies nicht der Idee von Agilität entsprach, aber die Unternehmen, mit denen ich arbeitete, hatten dicke Mauern zwischen Entwicklung und Betrieb. Und insgeheim war ich froh darüber.

Glücklicherweise waren andere in der agilen Community nicht so selbstgefällig. Sie arbeiteten daran, die Mauern zwischen Entwicklung und Betrieb einzureißen und später auch die Mauern, die den Bereich IT-Sicherheit abtrennen. Diese Bewegung wurde unter dem Namen *DevOps* bekannt. Sie wird auch *DevSecOps* genannt.

ANMERKUNG

Wie bei so vielen Dingen im agilen Ökosystem wurde der Begriff »DevOps« von wohlmeinenden Menschen mit falschen Annahmen verzerrt ... und von weniger wohlmeinenden Unternehmen, die versuchen, schnelles Geld zu machen. Hier verwende ich ihn im ursprünglichen Sinne des Wortes: enge Zusammenarbeit zwischen Entwicklung, Betrieb und IT-Sicherheit.

Manche Leute dehnen DevOps auf noch mehr Bereiche aus, mit Begriffen wie DevSecBizOps, DevSecBizDataOps oder sogar Dev<Everything>Ops. Damit schließt sich natürlich der Kreis zu funktionsübergreifenden, autonomen Teams, die über alle Fähigkeiten verfügen, die sie brauchen, um erfolgreich zu sein. Oder, Sie wissen schon ... Agilität.

DevOps überwindet die Grenzen zwischen Entwicklung, Betrieb und IT-Sicherheit und ermöglicht es Ihrem Team, Software zu erstellen, die sicherer und zuverlässiger ist und sich in der Produktion leichter verwalten lässt. Dieses Kapitel enthält vier Praktiken, die Sie dabei unterstützen:

- Abschnitt »Für den Betrieb bauen« auf Seite 589 zeigt, wie Software erstellt wird, die sicher und einfach in der Produktion zu verwalten ist.
- Abschnitt »Feature Flags« auf Seite 601 ermöglicht es Ihrem Team, unvollständige Software bereitzustellen.
- Abschnitt »Continuous Deployment« auf Seite 607 reduziert das Risiko und die Kosten der Produktionsbereitstellung.
- Abschnitt »Evolutionäre Systemarchitektur« auf Seite 614 hält Ihr System einfach, wartbar und flexibel.

Ursprünge von DevOps

Der Begriff »DevOps« wurde von Patrick Dubois geprägt, der 2009 die erste DevOpsDays-Konferenz veranstaltete. Die Idee, die Barrieren zwischen Entwicklung und Betrieb abzubauen, geht dem Begriff voraus, ohne dass es eine eindeutige Quelle gibt, aber eines der frühesten Beispiele ist Benjamin Treynor Sloss' Gründung des Site Reliability Engineering (SRE) bei Google im Jahr 2003. In [Beyer et al. 2016] schreibt er: »Man könnte DevOps als eine Verallgemeinerung mehrerer SRE-Kernprinzipien ansehen ... [oder] SRE als eine spezifische Implementierung von DevOps mit einigen eigenwilligen Erweiterungen betrachten.« Dieser Kerngedanke der Zusammenarbeit ist im Abschnitt *Für den Betrieb bauen* festgehalten .

Feature Flags sind auch als Feature Toggles bekannt. Wie DevOps sind sie eine relativ natürliche Erweiterung agiler Ideen – in diesem Fall von Continuous Integration – ohne eindeutige Quelle.

Continuous Deployment ist auch eine natürliche Erweiterung von Continuous Integration. Kent Beck hat eine ähnliche Praxis, »Daily Deployment«, in der zweiten Auflage seines XP-Buches aufgenommen [Beck 2004]. Die erste mir bekannte Verwendung des Begriffs findet sich in Paul Duvalls Buch »Continuous Integration« [Duvall 2006].

Evolutionäre Systemarchitektur ist eine Anwendung der evolutionären Entwurfsideen von XP auf die Systemarchitektur.

15.1 Für den Betrieb bauen

Zielgruppe

Entwicklerinnen, Betrieb

Unsere Software ist sicher und in der Produktion einfach zu verwalten.

Der Grundgedanke hinter DevOps ist einfach: Durch die Einbeziehung von Personen mit Betriebs- und Sicherheitskenntnissen in das Team können wir die Betriebstauglichkeit und Sicherheit in die Software einbauen, anstatt sie nachträglich hinzuzufügen. Das ist gemeint mit *Für den Betrieb bauen*.

Das ist wirklich alles, was es zu tun gibt! Beziehen Sie Personen mit Betriebs- und Sicherheitskenntnissen in Ihr Team ein oder beteiligen Sie sie zumindest an den Entscheidungen Ihres Teams. Lassen Sie sie an den Planungssitzungen teilnehmen. Erstellen Sie Storys, die die Überwachung, Verwaltung und Sicherheit Ihrer Software erleichtern. Diskutieren Sie, warum diese Storys wichtig sind, und setzen Sie entsprechende Prioritäten.

> Heben Sie Betriebs- und Sicherheitsstorys nicht für das Ende der Entwicklung auf.

Heben Sie Betriebs- und Sicherheitsstorys nicht für das Ende der Entwicklung auf. Es ist besser, Ihre Software jederzeit für das Release bereitzuhalten (siehe Abschnitt »Reduzieren Sie angefangene Arbeit« auf Seite 201). Wenn Sie Ihrer Software mehr Funktionen hinzufügen, sollten Sie auch die Betriebstauglichkeit entsprechend erweitern. Wenn Sie zum Beispiel eine Funktion hinzufügen, die eine neue Datenbank erfordert, fügen Sie auch Storys für die Bereitstellung, Überwachung, Sicherung und Wiederherstellung dieser Datenbank hinzu.

Welche Art von Betriebsabläufen und Sicherheitsanforderungen sollten Sie berücksichtigen? Das sollten Ihnen Ihre Teamkollegen sagen können. Die folgenden Abschnitte sollen Ihnen den Einstieg erleichtern.

Modellierung von Bedrohungen

Für den Betrieb zu bauen, bedeutet, dass Sie sich von Anfang an Gedanken über die Sicherheits- und Betriebsanforderungen machen, nicht erst am Ende der Entwicklung. Eine Möglichkeit, diese Anforderungen zu verstehen, ist die Modellierung von Bedrohungen. Dabei handelt es sich um eine Sicherheitstechnik, aber ihre Analyse ist auch für den Betrieb hilfreich.

Verwandte Themen

Visuelles Planen (S. 217),
Aufdecken blinder Flecken (S. 636)

Bei der *Bedrohungsmodellierung* geht es darum, Ihr Softwaresystem zu verstehen und herauszufinden, wie es kompromittiert werden kann. In seinem Buch *Threat Modeling: Designing for Security* [Shostack 2014] beschreibt Adam Shostack diesen Prozess als Beantwortung von vier Fragen. Es ist eine gute Teamaktivität:

Was bauen Sie? Stellen Sie Ihre Systemarchitektur dar: die Komponenten Ihrer eingesetzten Software, den Datenfluss zwischen den Komponenten und die Vertrauens- oder Autorisierungsbeziehungen zwischen ihnen.

1. *Was kann schiefgehen*? Nutzen Sie das gleichzeitige Brainstorming (siehe Abschnitt »Gleichzeitiges Arbeiten« auf Seite 119), um zu überlegen, wie jede Komponente und jeder Datenfluss angegriffen werden könnte, und grenzen Sie dann durch Dot Voting die größten Bedrohungen für Ihr Team ein.
2. *Was sollten Sie gegen die Dinge tun, die schiefgehen können*? Suchen Sie nach verschiedenen Möglichkeiten, wie Sie die größten Risiken überprüfen oder angehen können, und erstellen Sie dazu Storykarten. Fügen Sie diese Ihrem visuellen Plan hinzu.
3. *Haben Sie eine ordentliche Analyse durchgeführt*? Schlafen Sie eine Nacht darüber und sehen Sie sich Ihre Arbeit noch einmal an, um festzustellen, ob Sie etwas übersehen haben. Wiederholen Sie das Vorgehen regelmäßig, um neue Informationen und Erkenntnisse zu berücksichtigen. Nutzen Sie das Aufdecken blinder Flecken, um Lücken in Ihrem Denken aufzuspüren.

Weitere Informationen finden Sie in [Shostack 2014]. Das Buch ist für Personen ohne vorherige Erfahrung in IT-Sicherheit geschrieben und Kapitel 1 enthält alles, was Sie für den Anfang brauchen, einschließlich eines Kartenspiels zum Brainstorming von Bedrohungen. (Das Kartenspiel ist auch kostenlos online verfügbar unter *https://oreil.ly/CgeKn*.) Eine kürzere Einführung mit einer gut durchdachten Teamaktivität finden Sie in Jim Gumbleys Buch *A Guide to Threat Modelling for Developers* [Gumbley 2020].

Konfiguration

Laut der *Zwölf-Faktoren-App* (engl. *The Twelve-Factor App*) [Wiggins 2017] besteht die bereitgestellte Software aus einer Kombination von *Code* und *Konfiguration*. Konfiguration bedeutet in diesem Fall den Teil Ihrer Software, der für jede Umgebung unterschiedlich ist: Strings für Datenbankverbindungen, URLs und Passwörter für Dienste von Drittanbietern und so weiter.

Bei der Bereitstellung wird derselbe Code in jeder Umgebung eingesetzt, unabhängig davon, ob es sich um Ihren lokalen Rechner, eine Testumgebung oder die Produktionsumgebung handelt. Ihre Konfiguration wird sich jedoch ändern: Ihre Testumgebung wird beispielsweise für die Verwendung einer Testdatenbank konfiguriert, und die Konfiguration Ihrer Produktionsumgebung verwendet Ihre Produktionsdatenbank.

Diese Definition von »Konfiguration« umfasst *nur* Dinge, die sich zwischen den Umgebungen ändern. Teams machen oft andere Dinge konfigurierbar, wie z.B. das Copyright-Datum in der Fußzeile einer Website. Aber diese Arten der Konfiguration sollten klar von der Umgebungskonfiguration getrennt werden.

Sie sind Teil des Verhaltens Ihrer Software und sollten wie Code behandelt werden, einschließlich der Versionskontrolle neben Ihrem Code. Ich verwende zu diesem Zweck oft echten Code, z. B. Konstanten oder Getter in einem Konfigurationsmodul. (In der Regel programmiere ich dieses Modul auch, um die Umgebungskonfiguration zu abstrahieren.)

Die Umgebungskonfiguration hingegen sollte von Ihrem Code isoliert werden. Sie wird oft in einem separaten Repository gespeichert. Wenn Sie sie in Ihr Code-Repository aufnehmen, was sinnvoll ist, wenn Ihr Team für die Bereitstellung verantwortlich ist, halten Sie sie klar getrennt: zum Beispiel den Quellcode in einem Quellverzeichnis und die Umgebungskonfiguration in einem Umgebungsverzeichnis. Während der Bereitstellung wird dann die Konfiguration in die Bereitstellungsumgebung eingespeist, indem Umgebungsvariablen gesetzt, Dateien kopiert werden usw. Die Einzelheiten hängen von Ihrem Bereitstellungsmechanismus ab.

Verwandtes Thema

Feature Flags (S. 601)

Vermeiden Sie es, Ihre Software unendlich konfigurierbar zu machen. Komplizierte Konfigurationen sind am Ende eine Art Code, der in einer besonders schlechten Programmiersprache geschrieben ist, ohne Abstraktionen oder Tests. Wenn Sie eine ausgefeilte Konfigurierbarkeit benötigen, verwenden Sie stattdessen Feature Flags, um Verhalten selektiv ein- und auszuschalten. Wenn Sie komplexes kundengesteuertes Verhalten benötigen, ziehen Sie eine Plug-in-Architektur in Betracht. Bei beiden Ansätzen können Sie die Details in einer echten Programmiersprache programmieren.

Geheime Schlüssel

Teammitglieder sollten keinen Zugriff auf geheime Schlüssel haben.

Geheime Schlüssel – Passwörter, API-Schlüssel usw. – sind eine besondere Art der Konfiguration. Es ist besonders wichtig, dass sie nicht Teil Ihres Quellcodes sind. Tatsächlich sollten die meisten Teammitglieder überhaupt keinen Zugriff darauf haben. Definieren Sie stattdessen ein sicheres Verfahren zum Erzeugen, Speichern, Rotieren und Überprüfen von geheimen Schlüsseln. Bei komplexen Systemen ist dies oft mit einem Dienst oder Werkzeug zur Verwaltung von privaten Schlüsseln verbunden.

Wenn Sie die Umgebungskonfiguration in einem separaten Repository aufbewahren, können Sie den Zugriff auf die Geheimnisse kontrollieren, indem Sie den Zugriff auf das Repository streng einschränken. Wenn Sie die Umgebungskonfiguration in Ihrem Code-Repository aufbewahren, müssen Sie Ihre Geheimnisse »im Ruhezustand« verschlüsseln. Das bedeutet, Sie verschlüsseln alle Dateien, die Geheimnisse enthalten. Programmieren Sie Ihr Deployment-Skript so, dass es die Geheimnisse entschlüsselt, bevor es sie in die Bereitstellungsumgebung einbringt.

Apropos Bereitstellung: Achten Sie besonders darauf, wie Geheimnisse von Ihrer Build- und Deployment-Automatisierung verwaltet werden. Es ist zwar bequem, Geheimnisse in einem Deployment-Skript oder einer Konfiguration vom CI-Server abzulegen, aber dieser Komfort ist das Risiko nicht wert. Für die Automatisierung müssen dieselben sicheren Verfahren gelten wie für den restlichen Code.

Schreiben Sie niemals Geheimnisse in Ihre Logdateien. Da dies leicht versehentlich passieren kann, sollten Sie in Erwägung ziehen, Ihren Logging Wrapper so zu schreiben, dass er nach Daten sucht, die nach Geheimnissen aussehen (z. B. Felder mit dem Namen »Passwort« oder »geheim«), und sie unkenntlich macht. Wenn er einen solchen Fehler findet, sollte er einen Alarm auslösen, damit das Team den Fehler beheben kann.

Paranoia-Telemetrie

Ganz gleich, wie sorgfältig Sie Ihren Code programmieren, es wird in der Produktion trotzdem zu Fehlern kommen. Selbst perfekter Code hängt von einer unvollkommenen Welt ab. Externe Dienste geben unerwartete Daten zurück oder – noch schlimmer – reagieren sehr langsam. Dateisysteme haben keinen Platz mehr. Ausfallsicherungen ... fallen aus.

Jedes Mal, wenn Ihr Code mit der Außenwelt interagiert, programmieren Sie ihn so, dass er davon ausgeht, dass die Welt hinter Ihnen her ist. Überprüfen Sie jeden Fehlercode. Validieren Sie jede Antwort. Limitieren Sie bei nicht reagierenden Systemen die Wartezeiten. Programmieren Sie die Logik zum wiederholten Aufrufen so, dass sie exponentielle Wartezeiten verwendet.

Wenn Sie einem defekten System sicher aus dem Weg gehen können, tun Sie es. Wenn das nicht möglich ist, sollten Sie kontrolliert und sicher scheitern. In jedem Fall sollten Sie das aufgetretene Problem loggen, damit Ihr Überwachungssystem eine Warnung senden kann.

Logging

Niemand möchte mitten in der Nacht von einem Produktionsproblem geweckt werden. Aber trotzdem kommt es vor. Wenn es passiert, muss der Bereitschaftsdienst in der Lage sein, das Problem mit minimalem Aufwand zu diagnostizieren und zu beheben.

Um dies zu erleichtern, sollten Sie beim Logging systematisch und durchdacht vorgehen. Fügen Sie Log-Anweisungen nicht einfach überall in Ihrem Code hinzu. Überlegen Sie stattdessen, was fehlschlagen kann und was die Leute dazu wissen müssen. Erstellen Sie User Stories, um diese Fragen zu beantworten. Wenn Sie zum Beispiel eine Sicherheitslücke entdecken, wie werden Sie feststellen, welche Benutzerinnen und Daten betroffen waren? Wenn sich die Leistung verschlechtert, wie

werden Sie feststellen, was zu beheben ist? Eine vorausschauende Analyse (siehe Abschnitt »Vorausschauende Analyse« auf Seite 225) und Ihr Risikomodell können Ihnen helfen, diese Storys zu identifizieren und zu priorisieren.

Verwenden Sie *strukturierte Logs*, die an einen zentralen Datenspeicher weitergeleitet werden. Damit sind Ihre Protokolle leichter zu durchsuchen und zu filtern. Ein strukturiertes Log gibt Daten in einem maschinenlesbaren Format, z. B. JSON, aus. Schreiben Sie Ihren Logging Wrapper so, dass er das Logging beliebiger Objekte unterstützt. So können Sie problemlos Variablen einbinden, die wichtigen Kontext liefern.

Ich habe zum Beispiel an einem System gearbeitet, das von einem Dienst abhing, der seine veralteten APIs mit speziellen Antwort-Headern kennzeichnete. Wir haben unsere Software so programmiert, dass sie das Vorhandensein dieser Header überprüfte und diesen Node.js-Code ausführte:

```
log.action({
  code: "L16",
  message: "ExampleService-API ist veraltet.",
  endpoint,
  headers: response.headers,
});
```

Die Ausgabe sah wie folgt aus:

```
{
  "timestamp": 16206894860033,
  "date": "Mon May 10 2021 23:31:26 GMT",
  "alert": "Aktion",
  "component": "BeispielApp",
  "node": "web.2",
  "correlationId": "b7398565-3b2b-4d80-b8e3-4f41fdb21a98",
  "code": "L16",
  "message": "ExampleService-API ist veraltet.",
  "endpoint": "/v2/accounts/:account_id",
  "headers": {
    ⋮
    "x-api-version": "2.22",
    "x-deprecated": true,
    "x-sunset-date": "Wed November 10 2021 00:00:00 GMT"
  }
}
```

Durch den zusätzlichen Kontext, den die Protokollnachricht lieferte, war das Problem leicht zu diagnostizieren. Die Beschreibung unter `endpoint` machte deutlich, welche API genau veraltet war, und das `headers`-Objekt ermöglichte es den Entwicklerinnen, die Details zu verstehen und Fehlalarme auszuschließen.

Geben Sie auch beim Auslösen von Exceptions den Kontext an. Wenn Sie zum Beispiel eine `switch`-Anweisung haben, die niemals den Standardfall ausführen sollte, können Sie überprüfen, dass dieser nicht ausgeführt wird. Aber melden Sie nicht einfach »unerreichbaren Code ausgeführt«. Geben Sie eine detaillierte Exception an, die es Ihrem Team ermöglicht, das Problem zu beheben. Zum Bei-

spiel: »Unbekannter Benutzerabonnementtyp 'legacy-036' beim Versuch, eine Willkommens-E-Mail zu senden, aufgetreten.«

Im obigen Beispiel für das Logging sehen Sie mehrere Standardfelder. Die meisten von ihnen wurden von der Logging-Infrastruktur hinzugefügt. Ziehen Sie in Betracht, sie auch in Ihre Logdateien aufzunehmen:

- *Timestamp*: Maschinenlesbare Version des Zeitpunkts, zu dem der Logeintrag erstellt wurde.
- *Date*: Eine von Menschen lesbare Version vom `Timestamp`.
- *Alert*: Welche Art von Warnhinweis gesendet werden soll. Wird auch häufig »Stufe« genannt. Ich gehe gleich näher darauf ein.
- *Component*: Die Codebasis, die den Fehler erzeugt hat.
- *Node*: Der spezifische Rechner, der den Fehler verursacht hat.
- *Correlation ID*: Eine eindeutige ID, die zusammengehörige Logs zusammenfasst, und das oft über mehrere Dienste hinweg. Zum Beispiel können alle Protokolle, die sich auf eine einzige HTTP-Anfrage beziehen, dieselbe Correlation ID haben. Sie wird auch als »Request ID« bezeichnet.
- *Code*: Ein willkürlicher, eindeutiger Code für dieses Logging. Er ändert sich nie. Er kann zum Durchsuchen der Logdateien und zum Nachschlagen in der Dokumentation verwendet werden.
- *Message*: Eine von Menschen lesbare Version des `Codes`. Anders als `Code` kann sie sich ändern.

Ergänzen Sie Ihre Logs mit einer Dokumentation, die eine kurze Erläuterung zu jedem Warnhinweis enthält und, was noch wichtiger ist, was Sie dagegen tun können. Dies wird oft Teil Ihres *Handbuchs* sein, das eine Reihe von Verfahren und Prozessen für eine bestimmte Software enthält. Zum Beispiel:

L16: Die ExampleService-API ist veraltet.

Erläuterung: ExampleService, unser Abrechnungsdienstleister, plant, eine von uns verwendete API zu entfernen.

Warnhinweis: Wir müssen handeln. Unsere Anwendung wird nicht mehr funktionieren, wenn die API entfernt wird. Wir müssen uns also unbedingt darum kümmern. Aber es gibt eine Frist, sodass wir uns nicht sofort darum kümmern müssen.

Maßnahmen:

- Der Code muss wahrscheinlich auf die neue API aktualisiert werden. Wenn dies der Fall ist, sollte das `headers`-Objekt, das die vom ExampleService zurückgegebenen Header enthält, einen `x-deprecated`-Header und einen `x-sunset-date`-Header enthalten.

- Das Feld `endpoint` zeigt die spezifische API an, die den Alarm ausgelöst hat, aber auch andere von uns verwendete Endpunkte können betroffen sein.
- Die Dringlichkeit des Upgrades hängt davon ab, wann die API abgeschaltet wird. Das wird durch den `x-sunset-date`-Header angezeigt. Sie können dies überprüfen, indem Sie die Dokumentation vom ExampleService auf *example.com* lesen.
- Wahrscheinlich möchten Sie diese Warnmeldung deaktivieren, bis die API aktualisiert wurde. Achten Sie darauf, dass Sie nicht versehentlich gleichzeitig Warnungen für andere APIs deaktivieren, und überlegen Sie, die Warnmeldung automatisch wieder zu aktivieren, damit das Upgrade nicht in Vergessenheit gerät.

Beschreiben Sie, was der Leser *wissen muss*, nicht was er *tun muss*.

Achten Sie auf den lockeren, unbestimmten Ton des Abschnitts »Maßnahmen«. Das ist gewollt. Detaillierte Verfahren können zu einer »Zwickmühle von Verantwortung und Autorität« führen, bei der Menschen Angst haben, ein Vorgehen anzupassen, obwohl es nicht für ihre Situation passt [Woods et al. 2010, Kap. 8]. Indem Sie beschreiben, was der Leser *wissen muss*, nicht was er *tun muss*, wird die Entscheidungsbefugnis wieder in die Hände des Lesers bzw. der Leserin gelegt. Damit kann er oder sie die Ratschläge auf die jeweilige Situation anpassen.

Metriken und Beobachtbarkeit

Zusätzlich zum Logging sollte Ihr Code auch wichtige Elemente messen, d.h. sogenannte *Metriken* anlegen. Die meisten Metriken sind technischer Natur, wie z.B. die Anzahl der Anfragen, die Ihre Anwendung erhält, die Zeit, die für die Beantwortung von Anfragen benötigt wird, die Speichernutzung und so weiter. Einige Metriken werden jedoch fachlich orientiert sein, wie z.B. die Anzahl und der Wert der Kundenkäufe.

Metriken werden in der Regel über einen bestimmten Zeitraum hinweg gesammelt und dann gemeldet. Dies kann innerhalb Ihrer Anwendung über das Logging geschehen oder durch das Senden von Events an ein Werkzeug, das die Metriken aggregiert.

Gehen Sie mit Bedacht an die Beobachtbarkeit heran.

Zusammen erzeugen Ihre Protokolle und Metriken *Beobachtbarkeit*, d.h. die Fähigkeit, zu verstehen, wie sich Ihr System sowohl aus technischer als auch aus fachlicher Sicht verhält. Wie bei Ihren Protokollen sollten Sie auch bei der Beobachtbarkeit einen durchdachten Ansatz wählen.

Sprechen Sie mit Ihren Stakeholdern. Welche Beobachtungsfähigkeiten brauchen Sie aus *betrieblicher* Sicht? Aus einer *Sicherheitsperspektive*? Aus der *Geschäftsperspektive*? Aus der Perspektive des *Supports*? Erstellen Sie User Stories, um diesen Anforderungen gerecht zu werden.

Überwachung und Alarmierung

Die *Überwachung* erkennt, wenn Ihre Logs und Metriken Aufmerksamkeit erfordern. Ist dies der Fall, sendet das Überwachungswerkzeug eine *Warnmeldung* – eine E-Mail, eine Chat-Benachrichtigung oder sogar eine SMS oder einen Telefonanruf –, damit sich jemand um die Angelegenheit kümmern kann. In einigen Fällen erfolgt die Alarmierung durch einen separaten Dienst.

Die Entscheidungen darüber, wovor gewarnt werden soll und wovor nicht, können kompliziert sein. Daher kann Ihr Überwachungswerkzeug über eine geeignete Programmiersprache konfiguriert werden. Wenn das zutrifft, behandeln Sie diese Konfiguration mit dem Respekt, der einem echten Programm gebührt. Benutzen Sie eine Versionskontrolle, achten Sie auf den Entwurf und schreiben Sie Tests, wenn Sie können.

Welche Arten von Warnmeldungen Ihr Überwachungswerkzeug sendet, hängt von Ihrer Organisation ab, aber in der Regel gibt es vier Kategorien:

- *Emergency*: Irgendetwas brennt oder wird bald brennen und eine Person in Bereitschaft muss informiert werden und es *sofort* in Ordnung bringen.
- *Action*: Etwas Wichtiges erfordert Aufmerksamkeit, aber es ist nicht dringend genug, um jemanden zu alarmieren.
- *Monitor*: Etwas ist ungewöhnlich, und eine Person sollte einen zweiten Blick darauf werfen (verwenden Sie es sparsam).
- *Info*: Erfordert keine Aufmerksamkeit, ist aber nützlich für die Beobachtbarkeit.

Sie werden Ihr Überwachungswerkzeug so konfigurieren, dass es auf der Grundlage Ihres Loggings einige Warnmeldungen ausgibt. Obwohl es am einfachsten wäre, Ihre Logs so zu programmieren, dass sie die oben genannten Begriffe verwenden, benutzen die meisten Logbibliotheken stattdessen `FATAL`, `ERROR`, `WARN`, `INFO` und `DEBUG`. Obwohl sie technisch gesehen unterschiedliche Bedeutungen haben, können Sie sie direkt den vorhergehenden Stufen zuordnen: Verwenden Sie `FATAL` für Emergency, `ERROR` für Action, `WARN` für Monitor und `INFO` für ... Info. Verwenden Sie `DEBUG` überhaupt nicht, es verursacht nur zusätzliche Störgeräusche.

ANMERKUNG

Es ist in Ordnung, DEBUG-Logging während der Entwicklung zu verwenden. Aber checken Sie es nicht ein. Einige Teams programmieren ihr Continuous-Integration-Skript so, dass der Build automatisch fehlschlägt, wenn es DEBUG-Logging entdeckt.

Andere Warnmeldungen werden auf der Grundlage von Metriken ausgelöst. Sie werden in der Regel von Ihrem Überwachungswerkzeug erzeugt, nicht von Ihrem Anwendungscode. Achten Sie darauf, dass jede dieser Warnmeldungen einen Code zum Nachschlagen, eine von Menschen lesbare Meldung und eine Dokumentation in Ihrem Handbuch erhält, genau wie Ihre Logeinträge.

Verwandtes Thema

Continuous Deployment (S. 607)

Wenn möglich ziehe ich es vor, Entscheidungen zur Benachrichtigung in meinen Anwendungscode zu programmieren und Alarmierungen nicht durch meine Überwachungskonfiguration auszulösen, sondern über das Logging. So können die Teammitglieder die »Logik« der Alarmierung mit Code programmieren, mit dem sie vertraut sind. Der Nachteil ist, dass eine Änderung der Warnmeldungen eine erneute Bereitstellung der Software erfordert, sodass dieser Ansatz am besten funktioniert, wenn Ihr Team Continuous Deployment verwendet.

Hüten Sie sich vor *Alarmierungsmüdigkeit*. Denken Sie daran: Eine »Notfall«-Alarmierung *alarmiert eine Person in Bereitschaft*. Es braucht nicht viele Fehlalarme, damit die Menschen aufhören, auf Warnungen zu achten, insbesondere bei »Überwachungs«-Alarmierungen. Auf jeden Alarm muss reagiert werden: entweder um ihn zu beheben oder um zu verhindern, dass ein erneuter Fehlalarm auftritt. Wenn eine Alarmierung durchweg unangemessen für die Stufe ist – z. B. ein »Notfall«, um den man sich eigentlich erst morgen kümmern kann, oder eine »Überwachungs«-Alarmierung, die nie auf ein echtes Problem hinweist –, sollten Sie sie herabstufen oder nach Möglichkeiten suchen, sie zielgerichteter zu gestalten.

> Beheben Sie entweder den Alarm oder verhindern Sie, dass ein erneuter Fehlalarm auftritt.

Ähnlich verhält es sich mit Meldungen, die durchweg eine routinemäßige Reaktion ohne wirkliches Nachdenken erfordern, diese sollten automatisiert werden. Wandeln Sie die routinemäßige Antwort in Code um. Dies gilt insbesondere für »Überwachungs«-Warnmeldungen, die zu einer Müllhalde für lästige Belanglosigkeiten werden können. Es ist in Ordnung, eine »Überwachungs«-Warnmeldung zu erstellen, um Ihnen zu helfen, das Verhalten Ihres Systems zu verstehen,

aber sobald Sie dieses Verständnis haben, sollten Sie die Meldung zu einer »Aktion« oder einem »Notfall« erweitern, indem Sie die Warnung zielgerichteter gestalten.

Um die Warnmeldungen unter Kontrolle zu halten, sollten Sie Entwicklerinnen in Ihre Bereitschaftsdienste einbeziehen. Das hilft ihnen zu verstehen, welche Alarmierungen wichtig sind und welche nicht. Das wiederum führt zu einem Code, der bessere Entscheidungen bei der Alarmierung trifft.

ANMERKUNG

In einigen Unternehmen gibt es spezielle Betriebsteams, die die Systeme verwalten und sich um den Bereitschaftsdienst kümmern. Dies funktioniert am besten, wenn das Entwicklungsteam zunächst seine eigenen Systeme verwaltet. Bevor es die Verantwortung abgibt, muss es ein gewisses Maß an Stabilität nachweisen. Für weitere Informationen siehe [Kim et al. 2016, Kap. 15].

Schreiben Sie Tests für Ihr Logging und Ihre Alarmierungen. Sie sind für den Erfolg Ihrer Software ebenso wichtig wie die benutzungsseitigen Funktionen. Diese Tests können schwierig zu schreiben sein, da sie oft globale Zustände betreffen. Doch das ist ein behebbares Entwurfsproblem, wie im Abschnitt »Globale Zustände kontrollieren« auf Seite 523 beschrieben. Die Folgen 11–15 von [Shore 2020b] zeigen, wie wir das machen.

Cargo-Kult

Das DevOps-Team

Es ist Ihre erste Woche im Job, als Sie den Kopf in das Büro Ihres Chefs stecken. »Hey Walther, hast du einen Moment Zeit? Ich habe eine kurze Frage.« Er nickt freundlich und bittet Sie, sich zu setzen.

Sie machen es sich bequem. »Wer sind die DevOps-Leute in unserem Team? Ich habe ein Problem, das ich mit jemandem besprechen muss, aber ich konnte noch nicht herausfinden, wie wir DevOps betreiben.«

»Im Team?« Walther zieht eine Augenbraue hoch. »Nein, wir haben natürlich ein eigenes DevOps-Team. Und auch ein ProdOps- und eDataOps-Team. DevOps kümmert sich um Testumgebungen und CI-Tools, ProdOps um die Produktionsumgebung, und DataOps ist für Datenbanken und Schemata zuständig. Ist dir das noch nicht bekannt? Dein letztes Unternehmen muss ziemlich klein gewesen sein, um alle unter einen Hut zu bringen.«

→

Cargo-Kult

»Nein, wir ...« Sie schütteln den Kopf. »Es ist eine lange Geschichte. Es war eine andere Art von DevOps. Wie auch immer, kann ich mit den ProdOps-Leuten sprechen? Es ist möglich, dass die API, die ich verwende, eine 'Festplatte-voll'-Exception auslöst und ich möchte mich mit ihnen über eine angemessene Reaktion abstimmen.«

Walther saugt die Luft durch seine Zähne ein. »Ooooh, das würde ich nicht tun. Die sind ein bisschen kratzbürstig und super beschäftigt. Alles muss über ihr Ticketsystem laufen, und sie hassen es, wenn wir ihre Zeit verschwenden. Wann wird die ›Festplatte voll‹ überhaupt auftreten? Logge einfach die Exception und gebe null zurück. So machen wir das hier auch. Wir müssen eine Deadline einhalten.«

Sie holen tief Luft, um zu argumentieren, aber Walther hält die Hand auf. »Ich weiß, dass es anders ist, als du es gewohnt bist, aber wir sind ein großes Unternehmen. Du kannst nicht einfach herumlaufen und die Leute mit Belanglosigkeiten behelligen. Die Dinge müssen vorschriftsmäßig gemacht werden.«

Fragen

Wie finden wir Zeit für all das?

Wie die Rezensentin Sarah Horan Van Treese sagte: Sie haben keine Zeit, es nicht zu tun. »Meiner Erfahrung nach verschwenden Teams, die an einer Software arbeiten, die nicht ›für den Betrieb gebaut‹ ist, in der Regel sehr viel Zeit mit Dingen, die bei besserer Beobachtbarkeit ganz vermieden oder zumindest innerhalb von Minuten diagnostiziert und behoben werden könnten.« Kümmern Sie sich jetzt darum, oder verschwenden Sie später mehr Zeit mit der Brandbekämpfung.

Betriebliche und sicherheitstechnische Anforderungen können wie alles andere auch mit Storys geplant werden. Um sicherzustellen, dass diese Storys priorisiert werden, sollten Sie dafür sorgen, dass Personen mit Kenntnissen in diesen Bereichen an Ihren Planungssitzungen teilnehmen.

Verwandte Themen

Visuelles Planen (S. 217),
Keine Fehler (S. 626),
Freiraum (S. 305)

Behandeln Sie Alarmierungen wie Fehler: Sie sollten unerwartet auftreten und in dem Moment ernst genommen werden. Jeder Alarm sollte sofort adressiert werden. Das gilt auch für Fehlalarme: Wenn Sie wegen etwas alarmiert werden, das gar kein Problem ist, sollten Sie die Warnmeldung verbessern, damit die Wahrscheinlichkeit, dass erneut Alarm geschlagen wird, sinkt.

Qualitätsverbesserungen, wie z.B. die Anpassung der Empfindlichkeit eines Alarms oder die Verbesserung einer Logmeldung, können mithilfe des Freiraums Ihres Teams vorgenommen werden.

Was ist, wenn wir niemanden in unserem Team haben, der über Fähigkeiten im Bereich Betrieb oder IT-Sicherheit verfügt?

Nehmen Sie sich die Zeit, Ihre Betriebsabteilung schon zu einem frühen Zeitpunkt der Entwicklung zu kontaktieren. Lassen Sie sie wissen, dass Sie ihren Beitrag zu den betrieblichen Anforderungen und zur Alarmierung wünschen, und fragen Sie, wie Sie sich regelmäßig mit ihnen abstimmen können, um ihr Feedback zu erhalten. Ich habe die Erfahrung gemacht, dass die Personen in der Betriebsabteilung angenehm überrascht sind, wenn Entwicklungsteams sie bitten, sich einzubringen, *bevor* etwas in Brand gerät. Sie sind in der Regel sehr hilfsbereit.

Ich habe weniger Erfahrung mit der Einbindung von Sicherheitsfachleuten, was zum Teil daran liegt, dass sie seltener anzutreffen sind als Leute aus dem Betrieb. Doch der allgemeine Ansatz ist derselbe: Setzen Sie sich frühzeitig während der Entwicklung mit ihnen in Verbindung und besprechen Sie, wie Sie die Sicherheit einbauen können, anstatt sie erst nachträglich hinzuzufügen.

Voraussetzungen

Jedes Team kann für den Betrieb bauen. Die besten Ergebnisse erzielen Sie mit einem Team, das aus Mitgliedern mit Betriebs- und Sicherheitskenntnissen oder guten Beziehungen zu Personen mit diesen Kenntnissen besteht, und mit einer Führungskultur im Management, die vorausschauendes Handeln wertschätzt.

Verwandtes Thema

Komplettes Team (S. 94)

Indikatoren

Wenn Sie für den Betrieb bauen:

- hat Ihr Team potenzielle Sicherheitsrisiken berücksichtigt und behandelt;
- sind die Warnmeldungen gezielt und relevant;
- ist Ihr Unternehmen darauf vorbereitet, beim Auftreten von Produktionsproblemen darauf zu reagieren;
- ist Ihre Software widerstandsfähig und stabil.

Alternativen und Experimente

Bei dieser Praktik geht es letztlich um die Erkenntnis, dass »funktionierende Software« bedeutet, dass sie *in der Produktion eingesetzt* und nicht nur programmiert wird und sie bewusst so entwickelt wird, dass sie den Anforderungen der Produktion entspricht. Der beste Weg, dies zu erreichen, ist, Personen mit Betriebs- und Sicherheitsexpertise als Teil des Teams einzubeziehen.

Sie können gerne mit anderen Ansätzen experimentieren. Unternehmen verfügen oft nur über eine begrenzte Anzahl an Betriebs- und Sicherheitspersonal, sodass es schwierig sein kann, in jedem Team Personen mit diesen Fähigkeiten einzusetzen. Checklisten und automatische Überwachungsinstrumente sind ein gängiger Ersatz. Meiner Erfahrung nach sind sie aber nur ein schwacher Abklatsch. Ein besserer Ansatz ist die Schulung der Teammitglieder, um ihre Fähigkeiten zu entwickeln.

Versuchen Sie mit mindestens einem Team zusammenzuarbeiten, das einen echten DevOps-Ansatz verfolgt und in dem qualifizierte Leute tätig sind, bevor Sie experimentieren. Auf diese Weise wissen Sie, wie Ihre Experimente im Vergleich abschneiden.

Weiterführende Literatur

The Twelve-Factor App [Wiggins 2017] ist eine schöne, prägnante Einführung in die betrieblichen Anforderungen mit einer soliden Anleitung, wie Sie es angehen können.

The DevOps-Handbook [Kim et al. 2016] ist ein detaillierter Leitfaden für DevOps. In Teil IV, »The Technical Practices of Feedback«, werden ähnliche Themen wie in dieser Praktik behandelt.

The Phoenix Project [Kim et al. 2013] ist ein Roman über einen angeschlagenen IT-Manager, der lernt, DevOps in seinem Unternehmen einzuführen. Obwohl es nicht unbedingt um das Bauen für den Betrieb geht, ist es eine unterhaltsame Lektüre und eine gute Möglichkeit, mehr über die DevOps-Mentalität zu erfahren.

15.2 Feature Flags

Zielgruppe

Entwicklerinnen

Wir installieren und veröffentlichen unabhängig voneinander.

Für viele Teams ist die *Freigabe* ihrer Software dasselbe wie die *Bereitstellung* ihrer Software. Sie installieren einen Zweig ihres Code-Repositorys in die Produktion und alles in diesem Zweig wird freigegeben. Wenn es etwas gibt, das sie nicht freigeben wollen, speichern sie es in einem separaten Zweig.

Verwandte Themen

Continuous Integration (S. 489),
Continuous Deployment (S. 607)

Für Teams, die mit Continuous Integration und Deployment arbeiten, funktioniert das nicht. Abgesehen von kurzlebigen Entwicklungszweigen haben sie nur einen Zweig: ihren Integrationszweig. Es gibt keinen Ort, an dem sie unfertige Arbeit verstecken können.

Feature Flags, auch bekannt als *Feature Toggles*, lösen dieses Problem. Sie verbergen den Code programmatisch, anstatt Repository-Zweige zu verwenden. So können Teams unfertigen Code bereitstellen, ohne ihn freizugeben.

Feature Flags können auf verschiedene Weise programmiert werden. Einige können zur Laufzeit gesteuert werden, sodass neue Funktionen und Fähigkeiten freigegeben werden können, ohne dass die Software neu bereitgestellt werden muss. Dadurch wird die Freigabe in die Hände der Stakeholder gelegt und nicht in die der Entwicklerinnen. Sie können sogar so eingestellt werden, dass die Software nach und nach freigegeben wird oder dass die Freigabe auf bestimmte Benutzertypen beschränkt wird.

Schlusssteine

Streng genommen ist die einfachste Art von Feature Flags überhaupt kein Feature Flag. Kent Beck nennt es einen »Keystone« (dt. *Schlussstein*) [Beck 2004, Kap. 9]. Es ist ganz einfach: Wenn Sie an etwas Neuem arbeiten, verbinden Sie die Benutzungsoberfläche zuletzt. Das ist der Schlussstein. Solange der Schlussstein nicht gesetzt ist – solange die Benutzungsoberfläche nicht verknüpft ist –, wird niemand wissen, dass der neue Code existiert, weil es keine Möglichkeit gibt, auf ihn zuzugreifen.

Als ich zum Beispiel eine Website auf einen anderen Authentifizierungsdienst umstellte, implementierte ich zunächst einen Infrastruktur-Wrapper für den neuen Dienst. Den größten Teil dieser Arbeit konnte ich erledigen, ohne ihn mit der Login-Schaltfläche zu verbinden. Bis ich das getan hatte, wussten die Benutzerinnen nichts von der Änderung, weil die Login-Schaltfläche noch die alte Authentifizierungsinfrastruktur verwendete.

Verwandte Themen

Testgetriebene Entwicklung (S. 501), Schnelle, zuverlässige Tests (S. 520)

Das wirft die Frage auf: Wenn Sie Ihre Änderungen nicht sehen können, wie können Sie sie dann testen? Die Antwort lautet: testgetriebene Entwicklung und fokussierte Tests. Die testgetriebene Entwicklung ermöglicht es Ihnen, Ihre Arbeit zu überprüfen, ohne sie laufen zu sehen. Fokussierte bzw. eng gefasste Tests zielen auf bestimmte Funktionen ab, ohne dass diese mit dem Rest der Anwendung verbunden sein müssen.

Letztendlich möchten Sie natürlich sehen, wie Ihr Code ausgeführt wird, entweder für die Feinabstimmung der Benutzungsoberfläche (was testgetrieben schwierig sein kann), für die Überprüfung durch die Kundin oder einfach nur, um Ihre Arbeit zu überprüfen. Auch TDD ist nicht perfekt.

Entwerfen Sie Ihren neuen Code so, dass er mit einer einzigen Zeile »verknüpft« wird. Wenn Sie wollen, dass er ausgeführt wird, fügen Sie diese Zeile ein. Wenn Sie ihn integrieren müssen, bevor Sie ihn freigeben können, kommentieren

Sie diese Zeile aus. Wenn Sie bereit für die Freigabe sind, schreiben Sie den passenden Test und entfernen die Kommentierung der Zeile ein letztes Mal.

Schlusssteine müssen nicht unbedingt eine Benutzungsoberfläche beinhalten. Alles, was Ihre Arbeit vor den Kunden verbirgt, kann als Schlussstein verwendet werden. Ein Beispiel: Ein Team nutzte Continuous Deployment für eine Überarbeitung seiner Website. Das Team stellte die neue Website auf einem echten Produktionsserver bereit, der jedoch keine Produktionsanfragen empfing. Niemand außerhalb des Unternehmens konnte die neue Website sehen, bis das Team die Produktionsanfragen vom alten Server auf den neuen umstellte.

Schlusssteine sind meine bevorzugte Methode, um unvollständige Arbeiten zu verbergen. Sie sind einfach, geradlinig und erfordern keine besondere Wartung oder Arbeit am Entwurf.

Schlusssteine sind meine bevorzugte Methode, um unvollständige Arbeiten zu verbergen.

Feature Flags

Feature Flags dienen dem gleichen Zweck wie Schlusssteine, nur dass sie Code zur Steuerung der Sichtbarkeit verwenden – normalerweise eine einfache `if`-Anweisung – und keinen Kommentar.

Setzen wir das Beispiel der Authentifizierung fort. Denken Sie daran, dass ich meine neue Authentifizierungsinfrastruktur programmiert habe, ohne sie mit der Login-Schaltfläche zu verbinden. Bevor ich sie verknüpfen konnte, musste ich sie in der Produktion testen, da es komplizierte Wechselwirkungen zwischen dem Drittanbieterdienst und meinen Systemen gab. Ich wollte aber nicht, dass meine Benutzer die neue Anmeldung verwenden, bevor ich sie getestet hatte.

Ich habe dieses Dilemma mit einem Feature Flag gelöst. Die Benutzer sahen den alten Login, ich sah den neuen. Der Code funktionierte wie folgt (Node.js):

```
if (siteContext.useAuthOForAuthentication()) {
  // neues AuthO-HTML
}
else {
  // altes Persona-HTML
}
```

Im Rahmen der Änderung musste ich eine neue Seite zur E-Mail-Validierung einrichten. Sie war für bestehende Benutzer nicht sichtbar, aber es war immer noch möglich, die URL manuell einzugeben, also habe ich auch das Feature Flag verwendet, um sie umzuleiten:

```
httpGet(siteContext) {
  if (!siteContext.useAuthOForAuthentication()) return redirectToAccountPage();
  ⋮
}
```

Feature Flags sind echter Code. Sie benötigen die gleiche Aufmerksamkeit für Qualität wie der übrige Code. Die Seite zur E-Mail-Validierung hatte zum Beispiel diesen Test:

```
it("leitet zur Account Page um, wenn das AuthO-Feature-Flag ausgeschaltet ist",
function() {
 const siteContext = createContext({ authO: false });
 const response = httpGet(siteContext);
 assertRedirects(response, "/v3/account"));
});
```

Achten Sie darauf, Feature Flags zu entfernen, wenn sie nicht mehr benötigt werden. Das wird leicht vergessen, was einer der Gründe ist, warum ich Schlusssteine den Feature Flags vorziehe. Um sich daran zu erinnern, können Sie eine Erinnerung in Ihren Teamkalender oder eine »Flag entfernen«-Story in den Aufgabenplan Ihres Teams aufnehmen. Manche Teams programmieren ihren Code für die Flags so, dass er nach Ablauf des Verfallsdatums eine Warnung protokolliert oder Tests fehlschlagen.

Woher weiß Ihr Code, wann das Flag aktiviert ist? Mit anderen Worten, wo implementieren Sie Ihr Äquivalent von `useAuthOForAuthentication()`? Sie haben dafür mehrere Möglichkeiten.

Anwendungskonfiguration

Die Anwendungskonfiguration ist der einfachste Weg, um Ihre Feature Flags zu kontrollieren. Ihr Konfigurationscode kann den Status des Flags aus einer Konstante, einer Umgebungsvariablen, einer Datenbank oder was auch immer Sie bevorzugen beziehen. Eine Konstante ist am einfachsten und daher meine erste Wahl. Aber eine Umgebungsvariable oder eine Datenbank ermöglicht es Ihnen, das Flag auf einer Maschine-zu-Maschine-Basis zu aktivieren oder zu deaktivieren, was Ihnen erlaubt, inkrementelle Releases durchzuführen.

Personenbezogene Konfiguration

Wenn Sie Ihr Flag abhängig davon aktivieren wollen, wer angemeldet ist, machen Sie es zu einem Zugriffsrecht, das mit Ihrer Person oder der Kontoverwaltung verbunden ist. Zum Beispiel: `user.privileges.logsInWithAuthO()`. Damit können Sie inkrementelle Freigaben auf der Grundlage von Untergruppen von Personen durchführen und selektiv Funktionen zum Testen von Ideen freigeben.

Verwechseln Sie Feature Flags nicht mit der Zugriffskontrolle für Personen. Obwohl Feature Flags verwendet werden können, um ein Feature vor einer Person zu verbergen, sind sie eine Möglichkeit, neue Features, auf die Personen sonst Zugriff hätten, *vorübergehend* zu verbergen. Die Benutzungszugriffskontrolle hingegen dient dazu, Funktionen zu verbergen, auf die Personen *niemals* Zugriff haben sollten. Sie können beide mit personenbezogenen Rechten implementiert werden, sollten aber getrennt verwaltet werden.

ANMERKUNG

Feature Flags sind einfach zu implementieren, aber sie können kompliziert zu verwalten sein. Sobald Sie sich mit inkrementellen Versionen und Benutzersegmentierung beschäftigen, lohnt es sich, die vielen Werkzeuge und Dienste für deren Verwaltung zu prüfen.

Ein Feature Flag ist eine Möglichkeit, neue Features *vorübergehend* auszublenden.

Wenn Sie z.B. eine neue White-Label-Funktion für Ihre Unternehmenskunden erstellen, könnten Sie ein Feature Flag verwenden, um sie schrittweise für diese Kunden einzuführen. Sie würden aber zusätzlich noch eine Benutzungsberechtigung implementieren, die den Zugriff auf Unternehmenskunden einschränkt. Wenn der Code für das Feature Flag entfernt wird, dann haben nur noch Unternehmenskunden Zugriff und es besteht keine Gefahr, dass das Feature Flag versehentlich für die falschen Personen aktiviert wird.

Geheimnisse

In manchen Fällen möchten Sie ein Flag von Fall zu Fall aktivieren, können diese Berechtigung aber nicht einer Person zuweisen. Während meiner Authentifizierungsumstellung musste ich zum Beispiel die neue Login-Schaltfläche aktivieren, bevor ich tatsächlich angemeldet war.

In diesen Fällen können Sie ein Geheimnis verwenden, um das Flag zu aktivieren. In clientbasierten Anwendungen kann das Geheimnis eine spezielle Datei im Dateisystem sein. Bei serverbasierten Anwendungen kann ein Cookie oder ein anderer Request-Header verwendet werden. Genau das habe ich für mein Authentifizierungsflag getan. Ich habe den Code so programmiert, dass er nach einem verborgenen Cookie sucht, das nur gesetzt werden kann, wenn man sich als Administrator anmeldet.

Flags, die auf Geheimnissen basieren, sind riskanter als auf Konfigurationen basierende Flags. Wenn das Geheimnis bekannt wird, kann jede Person die Funktion aktivieren. Sie sind auch schwieriger einzurichten und zu kontrollieren. Ich verwende sie nur als letzte Möglichkeit.

Voraussetzungen

Jede Person kann Schlusssteine verwenden. Bei Feature Flags besteht die Gefahr, dass sie außer Kontrolle geraten. Deshalb muss Ihr Team auf deren Entwurf und Entfernen achten, insbesondere wenn sie sich vermehren. Collective Code Ownership und reflektierender Entwurf helfen dabei.

Verwandte Themen

Collective Code Ownership (S. 440),
Reflektierender Entwurf (S. 574)

Trotz ihrer oberflächlichen Ähnlichkeit mit Zugriffsrechten, die den Benutzungszugriff auf Funktionen steuern, sind Feature Flags nur als temporäre Steuerung gedacht. Verwenden Sie Feature Flags nicht als Ersatz für eine angemessene Benutzerzugriffskontrolle.

Indikatoren

Wenn Sie Schlusssteine und Feature Flags gut einsetzen:

- kann Ihr Team Software bereitstellen, die unvollständigen Code enthält;
- ist die Freigabe eine Business-Entscheidung, keine technische;
- ist der Code im Zusammenhang mit Flags sauber, gut konzipiert und gut getestet;
- werden Flags und der dazugehörige Code entfernt, nachdem das entsprechende Feature freigegeben wurde.

Alternativen und Experimente

Verwandte Themen

Refactoring (S. 529),
Reflektierender Entwurf (S. 574)

Feature Branches sind eine gängige Alternative zu Schlusssteinen und Feature Flags. Wenn eine Person mit der Arbeit an einer neuen Funktion beginnt, erstellt sie einen Zweig und fügt diesen erst dann wieder in den restlichen Code ein, wenn die Funktion fertiggestellt ist. Dies ist effektiv, um unfertige Änderungen vor den Kunden zu verbergen, aber größere Refactorings neigen dazu, Merge-Konflikte zu verursachen. Das macht es zu einer schlechten Wahl für *Delivering*-Teams, die auf Refactoring und reflektierenden Entwurf setzen, um die Kosten niedrig zu halten.

Schlusssteine sind so einfach, dass sie nicht viel Freiraum für Experimente lassen. Feature Flags hingegen sind ausgereift, um damit zu experimentieren. Suchen Sie nach Möglichkeiten, Ihre Feature Flags zu organisieren und den Entwurf sauber zu halten. Überlegen Sie, wie Ihre Flags neue Geschäftsfunktionen bieten können. Feature Flags werden zum Beispiel häufig für A/B-Tests verwendet, bei denen verschiedene Versionen Ihrer Software verschiedenen Benutzergruppen gezeigt werden, um dann auf der Grundlage der Ergebnisse Entscheidungen zu treffen.

Denken Sie beim Experimentieren daran: Weniger ist mehr. Obwohl Schlusssteine wie ein billiger Trick erscheinen mögen, sind sie sehr effektiv und halten den Code sauber. Es ist leicht möglich, dass Feature Flags außer Kontrolle geraten. Bleiben Sie deshalb bei einfachen Lösungen, wann immer Sie können.

Weiterführende Literatur

Martin Fowler geht in seinem Artikel »Keystone Interface« [Fowler 2020a] ausführlicher auf Schlusssteine ein.

Pete Hodgson diskutiert sehr gründlich über Feature Flags in seinem Artikel »Feature Toggles (aka Feature Flags)« [Hodgson 2017].

15.3 Continuous Deployment

Zielgruppe

Entwicklerinnen, Betrieb

Unser neuester Code ist in Produktion.

Wenn Sie Continuous Integration nutzen, hat Ihr Team den größten Teil des Risikos bei der Freigabe beseitigt. Bei richtiger Durchführung bedeutet Continuous Integration, dass Ihr Team jederzeit bereit ist, eine neue Version zu veröffentlichen. Sie haben Ihren Code getestet und Ihre Deployment-Skripte ausgeführt.

Verwandtes Thema

Continuous Integration (S. 489)

Eine Risikoquelle bleibt bestehen. Wenn Sie Ihre Software nicht auf echten Produktionsservern installieren, ist es möglich, dass Ihre Software in der Produktion nicht funktioniert. Unterschiede in der Umgebung, im Datenverkehr und in der Nutzung können zu Fehlern führen, selbst bei sorgfältigst getesteter Software.

Continuous Deployment reduziert dieses Risiko. Es folgt demselben Prinzip wie Continuous Integration: Durch die häufige Bereitstellung kleiner Teile verringern Sie das Risiko, dass eine große Änderung Probleme verursacht, und Sie können Probleme leicht finden und beheben, wenn sie auftreten.

Obwohl Continuous Deployment eine wertvolle Praktik für routinierte *Delivering*-Teams ist, ist sie optional. Wenn Ihr Team noch in der Entwicklung steckt, sollten Sie sich zunächst auf die anderen Praktiken konzentrieren. Die vollständige Einführung von Continuous Integration, einschließlich automatisierter Deployments in einer Testumgebung (von manchen als »Continuous Delivery« bezeichnet), wird Ihnen fast ebenso viel Nutzen bringen.

Verwendung von Continuous Deployment

Continuous Deployment ist nicht schwer, erfordert aber eine Reihe von Voraussetzungen:

- Erstellen Sie ein Deployment-Skript ohne Reibungsverluste und ohne Ausfallzeiten, das Ihren Code automatisch bereitstellt.

Verwandte Themen

Keine Reibungsverluste (S. 476),
Continuous Integration (S. 489),
Keine Fehler (S. 626),
Feature Flags (S. 601),
Für den Betrieb bauen (S. 589)

- Verwenden Sie Continuous Integration, um Ihren Code für die Freigabe bereitzuhalten.
- Verbessern Sie die Qualität so weit, dass Ihre Software ohne manuelle Tests bereitgestellt werden kann.
- Verwenden Sie Feature Flags oder Schlusssteine, um Bereitstellungen von Freigaben zu entkoppeln.
- Richten Sie eine Überwachung ein, um Ihr Team bei Bereitstellungsfehlern zu alarmieren.

Sobald diese Voraussetzungen erfüllt sind, müssen Sie nur noch die Bereitstellung in Ihrem Continuous-Integration-Skript ausführen.

Verwandtes Thema

Komplettes Team (S. 94)

Die Einzelheiten Ihres Deployment-Skripts hängen von Ihrem Unternehmen ab. Ihr Team sollte aus Personen mit Betriebskenntnissen bestehen, die wissen, was jeweils erforderlich ist. Wenn nicht, bitten Sie Ihre Betriebsabteilung um Hilfe. Wenn Sie auf sich allein gestellt sind, sind die Bücher *Continuous Delivery* [Humble & Farley 2010] und *The DevOps Handbook* [Kim et al. 2016] nützliche Wissensquellen.

Ihr Deployment-Skript muss zu 100 Prozent automatisiert sein. Die Bereitstellung erfolgt bei jeder Integration, d. h. mehrmals pro Tag und möglicherweise sogar mehrmals pro Stunde. Manuelle Schritte führen zu Verzögerungen und Fehlern.

Erkennung von Fehlern beim Deployment

Ihr Überwachungssystem sollte Sie warnen, wenn ein Deployment fehlschlägt. Dazu gehört zumindest die Überwachung auf eine Zunahme von Fehlern oder eine Abnahme der Leistung, aber Sie können auch Geschäftskennzahlen wie die Anmeldequote der Benutzer betrachten. Programmieren Sie Ihr Deployment-Skript so, dass es auch Fehler erkennt, wie z. B. Netzwerkausfälle während der Bereitstellung. Wenn das Deployment abgeschlossen ist, muss Ihr Deployment-Skript den bereitgestellten Commit mit »Erfolg« oder »Fehler« kennzeichnen.

Um die Auswirkungen von Fehlern zu verringern, können Sie das Deployment auf einer Untergruppe von Servern, den sogenannten *Canary-Servern*, durchführen und automatisch die Metriken der alten und der neuen Bereitstellung vergleichen. Wenn sie sich wesentlich unterscheiden, wird ein Alarm ausgelöst und das Deployment gestoppt. Bei Systemen mit vielen Produktionsservern können Sie auch mehrere Stufen von Canary-Servern einsetzen. So könnten Sie beispielsweise mit der Verteilung auf 10 % der Server beginnen, dann auf 50 % und schließlich auf alle.

Behebung von Fehlern beim Deployment

Einer der Vorteile von Continuous Deployment ist, dass es das Risiko der Bereitstellung verringert. Da jede Bereitstellung nur ein paar Stunden Entwicklungsarbeit bedeutet, sind die Auswirkungen in der Regel gering. Wenn etwas schiefgeht, kann die Änderung rückgängig gemacht werden, ohne dass der Rest des Systems beeinträchtigt wird.

Wenn beim Deployment etwas schiefgeht, brechen Sie sofort ab und sorgen Sie dafür, dass das gesamte Team sich auf die Behebung des Problems konzentriert. In der Regel bedeutet dies, dass die Bereitstellung zurückgenommen wird.

Unterbrechen Sie Ihre Entwicklungsarbeiten

Wenn ein Deployment fehlschlägt, *brechen Sie sofort alles ab*: Alle Teammitglieder stellen ihre Arbeit ein und konzentrieren sich auf die Behebung des Produktionsproblems. Dadurch wird verhindert, dass sich Fehler ansammeln.

Im Folgenden finden Sie eine Zusammenfassung der Schritte, die zur Behebung einer fehlgeschlagenen Bereitstellung erforderlich sind:

1. Unterbrechen Sie Ihre Entwicklungsarbeiten.
2. Machen Sie die Änderung der Produktionsumgebung rückgängig.
3. Machen Sie die bereitgestellten Änderungen im Code-Repository rückgängig. Integrieren Sie sie und stellen Sie den Stand bereit.
4. Erstellen Sie im Team Aufgaben zur Behebung des fehlerhaften Codes.
5. Beginnen Sie das Deployment von vorn.
6. Planen Sie nach der Bereitstellung des Fixes eine Sitzung zur Vorfallanalyse.

Ein Deployment rückgängig machen

Beginnen Sie mit der Wiederherstellung des Systems in seinen vorherigen Arbeitszustand. Dazu gehört in der Regel ein *Rollback*, bei dem der Code und die Konfiguration der vorherigen Version wiederhergestellt werden. Zu diesem Zweck können Sie jedes Deployment in einem Versionskontrollsystem oder einfach eine Kopie der letzten Bereitstellung aufbewahren.

Eine der einfachsten Möglichkeiten, ein Rollback zu ermöglichen, ist die Verwendung eines *Blue/Green Deployment*. Dazu erstellen Sie zwei Kopien Ihrer Produktionsumgebung, die Sie willkürlich mit »Blau« und »Grün« bezeichnen. Anschließend konfigurieren Sie Ihr System so, dass der Datenverkehr an eine der beiden Umgebungen geleitet wird. Bei jeder Bereitstellung wird zwischen den beiden Umgebungen hin- und hergeschwenkt, sodass Sie den Datenverkehr in die vorherige Umgebung zurückleiten können.

Wenn zum Beispiel »Blau« aktiv ist, leiten Sie den Datenverkehr zu »Grün« weiter. Wenn die Bereitstellung abgeschlossen ist, wird der Datenverkehr nicht mehr an »Blau«, sondern an »Grün« weitergeleitet. Wenn die Bereitstellung fehlschlägt, können Sie den Datenverkehr einfach wieder nach »Blau« leiten.

Gelegentlich schlägt das Rollback fehl. Dies kann ein Hinweis auf eine Beschädigung der Daten oder ein Konfigurationsproblem sein. In jedem Fall haben Sie alle Hände voll zu tun, bis das Problem gelöst ist. Das Buch *Site Reliability Engineering* [Beyer et al. 2016] enthält in den Kapiteln 12–14 praktische Anleitungen für die Reaktion auf solche Vorfälle.

Reparieren Sie das Deployment

Das Rollback des fehlerhaften Deployments löst in der Regel das unmittelbare Produktionsproblem, doch Ihr Team ist noch nicht fertig. Sie müssen das zugrunde liegende Problem beheben. Der erste Schritt besteht darin, Ihren Integrationszweig wieder in einen bekannt guten Zustand zu bringen. Versuchen Sie nicht, das Problem zu beheben, sondern nur, Ihren Code und die Produktionsumgebung wieder zu synchronisieren.

Beginnen Sie damit, die Änderungen im Code-Repository rückgängig zu machen, damit Ihr Integrationszweig mit dem Code übereinstimmt, der tatsächlich in der Produktion ist. Wenn Sie Merge-Commits in Git verwenden, können Sie einfach `git revert` für den Integrations-Commit ausführen. Verwenden Sie dann Ihren normalen Continuous-Integration-Prozess, um den zurückgesetzten Code zu integrieren und bereitzustellen.

Das Deployment des zurückgesetzten Codes sollte ohne Zwischenfälle erfolgen, da Sie denselben Code bereitstellen, der bereits ausgeführt wird. Es ist trotzdem wichtig, dies zu tun, weil es sicherstellt, dass Ihr *nächstes* Deployment von einem bekannt guten Zustand ausgeht. Wenn bei dieser zweiten Bereitstellung ebenfalls Probleme auftreten, lässt sich das Problem auf ein Problem des Deployments eingrenzen und nicht auf ein Problem in Ihrem Code.

Sobald sich Ihr System wieder in einem bekannt guten Zustand befindet, können Sie den zugrunde liegenden Fehler beheben. Erstellen Sie Aufgaben für die Fehlerbehebung – in der Regel werden die Personen, die das Problem ausgelöst haben, es beheben – und alle können wieder normal arbeiten. Wenn das Problem behoben ist, sollten Sie einen Termin zur Analyse des Vorfalls ansetzen, um zu erfahren, wie Sie diese Art von Bereitstellungsfehlern in Zukunft vermeiden können.

Verwandtes Thema

Vorfallanalyse (S. 644)

Alternative: Vorwärts korrigieren

Einige Teams führen kein Rollback durch, sondern *korrigieren vorwärts*. Sie nehmen eine schnelle Korrektur vor – möglicherweise durch Ausführen von `git revert` – und stellen das Projekt erneut bereit. Der Vorteil dieses Ansatzes ist, dass

Sie Probleme mit Ihrem normalen Deployment-Skript beheben können. Rollback-Skripte können veraltet sein und genau dann versagen, wenn Sie sie am dringendsten benötigen.

Andererseits sind Deployment-Skripte in der Regel langsam, selbst wenn Sie die Möglichkeit haben, Tests zu deaktivieren (was nicht unbedingt eine gute Idee ist). Ein gut ausgeführtes Rollback-Skript kann in ein paar Sekunden abgeschlossen sein. Die Behebung von Fehlern kann einige Minuten dauern. Während eines Ausfalls zählen diese Sekunden. Aus diesem Grund ziehe ich trotz der Nachteile ein Rollback vor.

Inkrementelle Freigaben

Verwandtes Thema

Feature Flags (S. 601)

Bei großen oder risikoreichen Änderungen sollten Sie den Code in der Produktion testen, bevor Sie ihn den Benutzern zugänglich machen. Dies ist vergleichbar mit einem Feature Flag, mit dem Unterschied, dass Sie den neuen Code tatsächlich ausführen werden. (Feature Flags verhindern in der Regel, dass der verborgene Code überhaupt ausgeführt wird.) Für zusätzliche Sicherheit können Sie die Funktion schrittweise freigeben, indem Sie jeweils nur eine Untergruppe von Benutzern aktivieren.

Im *DevOps-Handbuch* [Kim et al. 2016] wird dies als *Dark Launch* bezeichnet. In Kapitel 12 findet sich ein Beispiel dafür, wie Facebook diesen Ansatz zur Veröffentlichung von Facebook-Chat verwendet hat. Der Chat-Code wurde auf die Clients geladen und so programmiert, dass er unsichtbare Testnachrichten an den Backend-Service sendet, sodass Facebook den Code vor der Freigabe an die Kunden testen konnte.

Datenmigration

Datenbankänderungen können nicht rückgängig gemacht werden – zumindest nicht, ohne Datenverluste zu riskieren –, daher erfordert die Datenmigration besondere Sorgfalt. Sie ist vergleichbar mit der Durchführung einer inkrementellen Freigabe: Zuerst wird bereitgestellt, dann wird migriert. Das Ganze erfolgt in drei Schritten:

1. Stellen Sie Code bereit, der sowohl das neue als auch das alte Schema versteht. Stellen Sie den Datenmigrationscode gleichzeitig bereit.
2. Nach erfolgreichem Deployment führen Sie den Datenmigrationscode aus. Er kann manuell oder automatisch als Teil Ihres Deployment-Skripts gestartet werden.
3. Wenn die Migration abgeschlossen ist, entfernen Sie manuell den Code, der das alte Schema versteht, und stellen den restlichen Code dann erneut bereit.

Durch die Trennung von Datenmigration und Deployment kann jedes Deployment fehlschlagen und zurückgesetzt werden, ohne dass Daten verloren gehen. Die Migration erfolgt erst, nachdem sich der neue Code in der Produktion als stabil erwiesen hat. Das ist etwas komplizierter als die Migration von Daten während des Deployments, aber es ist sicherer und ermöglicht eine Bereitstellung ohne Ausfallzeiten.

Migrationen mit großen Datenmengen erfordern besondere Sorgfalt, da das Produktionssystem während der Migration der Daten verfügbar bleiben muss. Für diese Art von Migrationen sollten Sie Ihren Migrationscode so schreiben, dass er inkrementell arbeitet – aus Leistungsgründen möglicherweise mit einer Ratenlimitierung – und beide Schemata gleichzeitig verwendet. Wenn Sie beispielsweise Daten von einer Tabelle in eine andere verschieben, könnte Ihr Code beim Lesen und Aktualisieren von Daten beide Tabellen betrachten, aber nur Daten in die neue Tabelle einfügen.

Verwandte Themen

Aufgabenplanung (S. 265),
Visuelles Planen (S. 217)

Achten Sie nach Abschluss der Migration darauf, Ihren Code sauber zu halten, indem Sie den veralteten Code entfernen. Wenn die Migration mehr als ein paar Minuten in Anspruch nimmt, fügen Sie dem Aufgabenplan Ihres Teams eine Erinnerung hinzu. Bei sehr langwierigen Migrationen können Sie eine Erinnerung in Ihren Teamkalender aufnehmen oder eine Story »Datenmigration abschließen« in den visuellen Plan Ihres Teams einfügen.

Dieser dreistufige Migrationsprozess gilt für jede Änderung des externen Zustands. Dazu gehören nicht nur Datenbanken, sondern auch Konfigurationseinstellungen, Änderungen an der Infrastruktur und an Diensten von Drittanbietern. Seien Sie sehr vorsichtig, wenn es um externe Zustände geht, denn diese Fehler lassen sich nur schwer rückgängig machen. Kleinere, häufigere Änderungen sind in der Regel besser als große, unregelmäßige Änderungen.

Voraussetzungen

Verwandte Themen

Continuous Integration (S. 489),
Feature Flags (S. 601),
Keine Fehler (S. 626),
Keine Reibungsverluste (S. 476),
Für den Betrieb bauen (S. 589)

Für Continuous Deployment benötigt Ihr Team einen rigorosen Continuous-Integration-Ansatz. Sie müssen mehrmals am Tag integrieren und jedes Mal ein bewährtes, einsatzbereites Artefakt erstellen. »Bereitstellen« bedeutet in diesem Fall, dass unfertige Funktionen vor den Benutzerinnen verborgen sind und Ihr Code nicht manuell getestet werden muss. Schließlich muss Ihr Deployment-Prozess vollständig automatisiert sein und Sie benötigen eine Möglichkeit, Fehler beim Deployment automatisch zu erkennen.

Continuous Deployment ist nur dann sinnvoll, wenn die Bereitstellungen für die Benutzerinnen unsichtbar sind. In der Praxis bedeutet das in der Regel Backend-Systeme und webbasierte Frontends. Desktop- und mobile Frontends, eingebettete Systeme usw. eignen sich normalerweise nicht für Continuous Deployment.

Indikatoren

Wenn Ihr Team kontinuierlich bereitstellt:

- wird das Deployment für die Produktion zu einem stressfreien Ereignis;
- lassen sich Probleme beim Deployment leicht beheben;
- ist es unwahrscheinlich, dass das Deployment Probleme in der Produktion verursacht und wenn doch, sind sie in der Regel schnell zu beheben.

Alternativen und Experimente

Die typische Alternative zum Continuous Deployment ist das *Release-orientierte Deployment*: Die Bereitstellung findet nur dann statt, wenn Sie etwas fertig haben, das Sie veröffentlichen können. Continuous Deployment ist tatsächlich sicherer und zuverlässiger, wenn die Voraussetzungen dafür gegeben sind, auch wenn es zunächst beängstigender klingt.

Sie müssen nicht direkt vom Release-orientierten Deployment zum Continuous Deployment wechseln. Sie können es langsam angehen, indem Sie zunächst ein vollständig automatisiertes Deployment-Skript schreiben, dann im Rahmen von Continuous Integration automatisch in einer Abnahmeumgebung bereitstellen und schließlich zum Continuous Deployment übergehen.

Was das Experimentieren angeht, so besteht der Kerngedanke von Continuous Deployment darin, die unfertige Arbeit zu minimieren und die Feedbackschleife zu beschleunigen (siehe den Abschnitt »Reduzieren Sie angefangene Arbeit« auf Seite 201 und den Kasten »Schnelles Feedback« auf Seite 503). Alles, was Sie tun können, um diese Feedbackschleife zu beschleunigen und die für das Deployment benötigte Zeit zu verkürzen, geht in die richtige Richtung. Wenn Sie nach Möglichkeiten suchen, die Feedbackschleife auch für *neue* Ideen zu beschleunigen, erhalten Sie zusätzliche Punkte.

Weiterführende Literatur

Das *DevOps-Handbuch* [Kim et al. 2016] bietet einen umfassenden Überblick über alle Aspekte von DevOps, einschließlich Continuous Deployment, mit einer Fülle von Fallstudien und Beispielen aus der Praxis.

Migrating bajillions of database records at Stripe [Heaton 2015] beinhaltet ein interessantes und unterhaltsames Beispiel für inkrementelle Datenmigration.

15.4 Evolutionäre Systemarchitektur

Zielgruppe

Entwicklerinnen, Betrieb

Wir bauen unsere Infrastruktur für das, was wir heute brauchen, ohne auf morgen zu verzichten.

Verwandte Themen

Einfacher Entwurf (S. 563), Inkrementeller Entwurf (S. 550), Reflektierender Entwurf (S. 574)

Einfachheit ist das Herzstück von Agilität, wie im Abschnitt »Einfachheit« auf Seite 563 beschrieben. Besonders deutlich wird dies in der Art und Weise, wie routinierte *Delivering*-Teams an den evolutionären Entwurf herangehen: Sie beginnen mit einem möglichst einfachen Entwurf, bauen darauf mit inkrementellem Entwurf mehr Funktionen auf und verfeinern und verbessern ihren Code ständig mit reflektierendem Entwurf.

Wie sieht es mit Ihrer Systemarchitektur aus? Mit *Systemarchitektur* meine ich die Komponenten, aus denen Ihr bereitgestelltes System besteht. Dazu gehören die von Ihrem Team erstellten Anwendungen und Dienste und die Art und Weise, wie sie interagieren, Ihre Netzwerk-Gateways und Lastverteiler und sogar Dienste von Drittanbietern. Was ist mit *ihnen*? Können Sie mit einfachen Mitteln starten und sich von dort aus weiterentwickeln?

Das ist *evolutionäre Systemarchitektur* und ich habe gesehen, dass sie bei kleinen Systemen funktioniert. Aber Systemarchitekturen entwickeln sich nur langsam weiter, sodass hinter der evolutionären Systemarchitektur nicht dieselbe umfassende Branchenerfahrung steht wie hinter dem evolutionären Entwurf. Entscheiden Sie selbst, wie und wann sie angewendet werden sollte.

ANMERKUNG

Ich unterscheide zwischen Systemarchitektur und Anwendungsarchitektur. Die *Anwendungsarchitektur* ist der Entwurf Ihres Codes, einschließlich der Entscheidungen darüber, wie andere Komponenten aus Ihrem System aufgerufen werden sollen. Sie wird im Abschnitt »Anwendungsarchitektur« auf Seite 555 behandelt. In dieser Praktik geht es um die *Systemarchitektur*: Entscheidungen darüber, *welche* Komponenten erstellt und verwendet werden sollen, und ihre Beziehungen auf hoher Ebene zueinander.

Werden Sie es wirklich brauchen?

Die Softwarebranche ist voll von Geschichten über große Unternehmen, die große Probleme lösen. Google hat eine Datenbank, die auf der ganzen Welt repli-

ziert wird! Netflix schloss seine Rechenzentren und verlagerte alles in die Cloud! Amazon verlangte, dass jedes Team seine Dienste veröffentlicht und schuf damit das Milliardengeschäft Amazon Web Services!

Es ist verlockend, diese Erfolgsgeschichten nachzuahmen, aber die Probleme, die diese Unternehmen gelöst haben, sind wahrscheinlich nicht die Probleme, die Sie lösen müssen. Bis Sie die Größe von Google, Netflix oder Amazon erreicht haben ... YAGNI (You aren't gonna need it). Sie werden es nicht brauchen.

Nehmen wir Stack Overflow, die beliebte Website für Fragen und Antworten rund um Programmierung. Dort werden 1,3 *Milliarden* Seiten pro Monat aufgerufen, wobei jede Seite in weniger als 19 ms erstellt wird. Wie machen die das?[1]

- 2 HAProxy-Load-Balancer, einer aktiv im Betrieb und einer zur Ausfallsicherung, mit Spitzenwerten von 4.500 Anfragen pro Sekunde und 18 % CPU-Auslastung
- 9 IIS-Webserver mit Spitzenwerten von 450 Anfragen pro Sekunde und 12 % CPU-Auslastung
- 2 Redis-Caches, ein Master und ein Replikat, mit Spitzenwerten von 60.000 Operationen pro Sekunde und 2 % CPU-Auslastung
- 2 SQL-Server-Datenbankserver für Stack Overflow, ein Live- und ein Standby-Server, mit Spitzenwerten von 11.000 Abfragen pro Sekunde und 15 % CPU-Auslastung
- 2 weitere SQL-Server-Instanzen für die anderen Stack Exchange-Websites, ein Live- und ein Standby-Server, mit einer Spitzenleistung von 12.800 Abfragen pro Sekunde und 14 % CPU-Auslastung
- 3 Tag-Engine- und ElasticSearch-Server mit durchschnittlich 3.644 Anfragen pro Minute und 3 % CPU-Auslastung für die benutzerdefinierte Tag-Engine und 34 Millionen Suchanfragen pro Tag und 7 % Auslastung für ElasticSearch
- 1 SQL-Server-Datenbankserver für die Protokollierung von HTTP-Anfragen
- 6 LogStash-Server für alle anderen Protokolle
- Ungefähr das Gleiche noch einmal in einem redundanten Rechenzentrum (für die Wiederherstellung im Notfall)

Im Jahr 2016 wurde die »Stack Overflow«-Website 5- bis 10-mal pro Tag ausgeliefert. Das vollständige Deployment dauerte etwa acht Minuten. Abgesehen von der Lokalisierung und der Datenbankmigration bestand die Bereitstellung darin, die Server in einer Schleife zu durchlaufen, jeden einzelnen aus der HAProxy-

1. Stack Overflow veröffentlicht die eigene Systemarchitektur und Leistungsstatistiken (*https://stackexchange.com/performance*), und Nick Craver hat eine ausführliche Artikelserie über ihre Architektur unter [Craver 2016] herausgebracht. Die zitierten Daten wurden am 4. Mai 2021 abgerufen.

Rotation zu nehmen, die Dateien zu kopieren und ihn wieder in die Rotation aufzunehmen. Die Hauptanwendung ist ein einzelner mandantenfähiger Monolith, der alle Fragen- und Antwort-Webseiten bedient.

Dies ist ein ausgesprochen unzeitgemäßer Ansatz für die Systemarchitektur. Es gibt keine Container, keine Microservices, keine automatische Skalierung, nicht einmal eine Cloud-Anbindung. Es gibt nur ein paar kräftige Server im Rack, eine Handvoll Anwendungen und ein Deployment, das Dateien kopiert. Das ist unkompliziert, robust und funktioniert.

Einer der häufigsten Gründe für eine komplexe Architektur ist die Skalierung. Aber Stack Overflow ist eine der 50 meistbesuchten Websites der Welt.[2] Ist Ihre Architektur komplexer als die von Stack Overflow? Wenn ja ... muss sie das sein?

Streben Sie nach Einfachheit

Das heißt nicht, dass Sie die Architektur von Stack Overflow blindlings kopieren sollten. Kopieren Sie nie etwas blindlings! Schauen Sie sich stattdessen die Probleme an, die *Sie* lösen müssen. (»Ein beeindruckenderer Lebenslauf« zählt nicht.) Was ist die einfachste Architektur, die diese Probleme lösen kann?

Eine Möglichkeit, sich dieser Frage zu nähern, besteht darin, mit einer idealisierten Sicht der Welt zu beginnen. Sie können dieses Gedankenexperiment sowohl für bestehende als auch für neue Architekturen verwenden.

1. Beginnen Sie mit einer idealen Welt

Stellen Sie sich vor, Ihre Komponenten sind auf magische Weise perfekt, aber nicht sofort programmiert. Das Netz ist absolut zuverlässig, aber es gibt immer noch eine Latenzzeit im Netz. Jeder Knoten verfügt über unbegrenzte Ressourcen. Wo würden Sie die Grenzen Ihrer Komponenten setzen?

Möglicherweise müssen Sie die Komponenten aus Gründen der Sicherheit, der Vorschriften und der Latenzzeit auf separate Server oder geografische Bereiche aufteilen. Wahrscheinlich werden Sie zwischen der clientseitigen und der serverseitigen Verarbeitung unterscheiden. Durch den Einsatz von Komponenten von Drittanbietern sparen Sie noch Zeit und Aufwand.

2. Einführung von unvollkommenen Komponenten und Netzen

Streichen Sie jetzt die Annahme, die Komponenten und Netze seien perfekt. Komponenten fallen aus, Netze brechen zusammen. Jetzt brauchen Sie Redundanz, was Komponenten für die Replikation und Ausfallsicherung erfordert. Wie können Sie diese Anforderungen am einfachsten erfüllen? Können Sie die Kom-

2. Traffic-Ranking von Stack Overflow abgerufen von *alexa.com* am 6. Mai 2021.

plexität reduzieren, indem Sie ein Werkzeug oder einen Dienst eines Drittanbieters verwenden? Stack Overflow muss sich zum Beispiel um redundante Stromversorgungen und Generatoren kümmern. Wenn Sie einen Cloud-Anbieter nutzen, ist das dessen Problem, nicht Ihres.

3. Ressourcen begrenzen

Als Nächstes streichen Sie die Annahme, dass die Ressourcen unbegrenzt sind. Möglicherweise benötigen Sie mehrere Knoten, um die Last zu bewältigen, sowie Komponenten für den Lastausgleich. Möglicherweise müssen Sie eine CPU-intensive Operation in eine eigene Komponente ausgliedern und eine Queue einführen, um sie zu beliefern. Möglicherweise benötigen Sie einen gemeinsamen Cache und eine Möglichkeit, ihn zu füllen.

Spekulieren Sie über die zukünftige Auslastung oder befassen Sie sich mit realen Problemen?

Aber seien Sie vorsichtig: *Spekulieren* Sie über die zukünftige Auslastung oder befassen Sie sich mit realen Problemen auf der Grundlage der tatsächlichen Nutzung und Trends? Können Sie Ihre Architektur vereinfachen, indem Sie leistungsfähigere Hardware einsetzen oder indem Sie warten, um zukünftige Belastungen zu adressieren?

4. Menschen und Teams berücksichtigen

Schließlich sollten Sie die idealisierte Programmierung entfernen. Wer wird die einzelnen Komponenten bauen? Wie werden sie sich untereinander abstimmen? Müssen Sie die Komponenten aufteilen, um die teamübergreifende Kommunikation zu erleichtern oder um die Komplexität einer einzelnen Komponente zu begrenzen? Überlegen Sie, wie Sie auch diese Einschränkungen vereinfachen können.

Kontrolle der Komplexität

Eine gewisse architektonische Komplexität ist notwendig. Auch wenn Ihr System einfacher wäre, wenn Sie sich keine Gedanken über Lastausgleich oder Komponentenausfall machen müssten, müssen Sie sich über diese Dinge Gedanken machen. Fred Brooks sagte in seinem berühmten Aufsatz »No Silver Bullet: Essence and Accident in Software Engineering« [Brooks 1995], dass eine gewisse Komplexität *unerlässlich* ist. Sie kann nicht eliminiert werden.

Andere Komplexität ist jedoch *unbeabsichtigt*. Manchmal teilt man eine große Komponente in mehrere kleine Komponenten auf, nur um sie für die menschliche Seite zu vereinfachen, und nicht, weil dies ein wesentlicher Teil des zu lösenden Problems ist. Zufällige Komplexität *kann* beseitigt oder zumindest reduziert werden.

Evolutionärer Entwurf

Einer der häufigsten Gründe für die Aufteilung von Komponenten ist die Vermeidung von »Big Ball of Mud«. Kleine Komponenten sind einfach und leicht zu warten.

> Kleine Komponenten erhöhen tendenziell die Gesamtkomplexität.

Leider wird dadurch die Komplexität nicht *verringert*, sondern lediglich von der Anwendungsarchitektur in die Systemarchitektur *verlagert*. Tatsächlich führt die Aufteilung einer großen Komponente in mehrere kleine Komponenten eher zu einer *Erhöhung* der Gesamtkomplexität. Einzelne Komponenten sind dann zwar leichter zu verstehen, aber die komponentenübergreifenden Interaktionen verschlechtern sich. Die Fehlerverfolgung ist schwieriger. Refactoring ist schwieriger. Und verteilte Transaktionen ... nun, die vermeidet man am besten ganz.

Sie können die Notwendigkeit, eine Komponente aufzuteilen, verringern, indem Sie einen evolutionären Entwurf verwenden. So können Sie große Komponenten erstellen, die keine »Big Ball of Mud« sind.

Verwandte Themen

Einfacher Entwurf (S. 563),
Inkrementeller Entwurf (S. 550),
Reflektierender Entwurf (S. 574)

Selbstdisziplin

Ein weiterer Grund, warum Teams ihre Komponenten aufteilen, ist die Entkopplung. Wenn eine Komponente für mehrere Datentypen verantwortlich ist, ist es verlockend, die Daten miteinander zu vermischen, was ein späteres Refactoring erschwert.

Natürlich gibt es keinen bestimmten Grund dafür, dass Daten miteinander verwoben sein müssen. Es ist lediglich eine Entwurfsentscheidung. Wenn Sie getrennte Komponenten entwerfen können, können Sie auch getrennte Module innerhalb einer einzigen Komponente entwerfen. Sie können sogar für jedes Modul eine eigene Datenbank verwenden. Es ist ja nicht so, dass durch Aufrufe über das Netzwerk auf magische Weise ein guter Entwurf entsteht!

Aber Aufrufe über das Netzwerk erzwingen eine Aufteilung. Wenn Sie das Netz nicht nutzen, um die Trennung zu erzwingen, brauchen Sie stattdessen ein Team mit Selbstdisziplin. Collective Code Ownership, Pair oder Mob Programming und energiegeladene Arbeit sind hier hilfreich, und ein reflektierender Entwurf ermöglicht es Ihnen, alle Fehler zu beheben, die Ihnen unterlaufen.

Verwandte Themen

Collective Code Ownership (S. 440),
Pair Programming (S. 448),
Mob Programming (S. 461),
Energiegeladene Arbeit (S. 176),
Reflektierender Entwurf (S. 574)

Schnelles Deployment

Große Komponenten sind oft schwierig bereitzustellen. Meiner Erfahrung nach ist nicht das Deployment selbst schwierig, sondern der *Build* und die *Tests*, die vor dem Deployment der Komponente durchgeführt werden müssen. Dies gilt umso mehr, wenn die Komponente manuell getestet werden muss.

Gehen Sie dieses Problem an, indem Sie einen reibungslosen Build erstellen, testgetriebene Entwicklung und Continuous Integration einführen und schnelle, zuverlässige Tests schreiben. Wenn Ihr Build und Ihre Tests schnell sind, müssen Sie eine Komponente nicht aufteilen, nur um das Deployment zu vereinfachen.

Verwandte Themen

Keine Reibungsverluste (S. 476), Testgetriebene Entwicklung (S. 501), Continuous Integration (S. 489), Schnelle, zuverlässige Tests (S. 520)

Vertikale Skalierung

Um Conways Gesetz zu paraphrasieren, neigen Organisationen dazu, ihre Organigramme auszuliefern. Viele Unternehmen tendieren zu horizontaler Skalierung (siehe Kap. 6), was zu vielen kleinen, isolierten Teams führt. Dementsprechend brauchen sie kleine Komponenten.

Die vertikale Skalierung ermöglicht es Ihren Teams, gemeinsam an denselben Komponenten zu arbeiten. So können Sie Ihre Architektur auf das zu lösende Problem abstimmen, anstatt sie passend zu Ihren Teams zu gestalten.

Refactoring der Systemarchitektur

Ich habe einen Freund, der in einem großen, bekannten Unternehmen arbeitet. Aufgrund eines architektonischen Top-down-Mandats unterhält sein Team von drei Programmierern 21 unterschiedliche Dienste – einen für jede von ihnen kontrollierte Entität. Einundzwanzig! Wir nahmen uns etwas Zeit, um darüber nachzudenken, wie der Code seines Teams vereinfacht werden könnte.

- Ursprünglich war sein Team verpflichtet, jeden Dienst in einem separaten Git-Repository zu speichern. Das Team erhielt die Erlaubnis, die Dienste in einem einzelnen Repository zusammenzufassen. Dadurch konnte das Team doppelten Serialisierungs-/Deserialisierungscode einsparen und Refactorings drastisch vereinfachen. Zuvor konnte eine einzige Änderung zu 16 separaten Commits in 16 Repositories führen. Jetzt braucht es nur noch einen.
- Bis auf wenige Ausnahmen sind die CPU-Anforderungen der Dienste seines Teams minimal. Dank eines unternehmensweiten Service-Locators könnten die Dienste zu einer einzigen Komponente zusammengefasst werden, ohne ihre Endpunkte zu ändern. Dadurch könnten sie auf weniger VMs bereitgestellt werden, was die Cloud-Kosten senken würde; Netzwerkaufrufe könnten

durch Funktionsaufrufe ersetzt werden, was die Antwortzeiten beschleunigen würde; und die Ende-zu-Ende-Tests könnten vereinfacht werden, was die Bereitstellung einfacher und schneller machen würde.

- Etwa die Hälfte der Dienste seines Teams wird nur innerhalb seines Teams genutzt. Jeder Dienst ist mit einer gewissen Menge an zusätzlichem Aufwand verbunden. Dieser Aufwand könnte beseitigt werden, wenn die internen Dienste in Bibliotheken umgewandelt würden. Damit würde auch eine Reihe langsamer Ende-zu-Ende-Tests entfallen.

Alles in allem könnte sein Team durch die Vereinfachung der Systemarchitektur eine Menge Kosten und Reibungsverluste bei der Entwicklung einsparen, wenn die Teammitglieder die Erlaubnis dazu erhalten würden.

Ich kann mir mehrere solcher Refactorings auf Systemebene vorstellen. Leider haben sie noch nicht die lange Historie, die die übrigen Ideen in diesem Buch aufweisen. Die »Mikrolith«-Refactorings sind besonders unerprobt. Daher werde ich sie nur kurz, ohne viele Details skizzieren. Betrachten Sie sie als eine Reihe von Ideen, die Sie berücksichtigen sollten, nicht als Kochbuch, dem Sie folgen müssen.

Multirepo-Komponenten → Monorepo-Komponenten

Wenn sich die Komponenten Ihres Teams in mehreren Repositorys befinden, können Sie sie in einem einzigen Repository zusammenfassen und den gemeinsamen Code für gemeinsame Typen und Dienste herausziehen.

Komponenten → Mikrolithen

Wenn Ihr Team über mehrere Komponenten verfügt, können Sie diese zu einer einzigen Komponente zusammenfassen und dabei die grundlegende Architektur beibehalten. Teilen Sie sie in separate Verzeichnisse auf und verwenden Sie eine Top-Level-Schnittstellendatei anstatt eines Servers, um zwischen Ihren serialisierten Daten im Aufruf und den Datenstrukturen der Komponente zu übersetzen. Ersetzen Sie die Netzwerkaufrufe zwischen den Komponenten durch Funktionsaufrufe, aber behalten Sie die Architektur in jeder anderen Hinsicht bei, einschließlich der Verwendung von primitiven Datentypen anstelle von Objekten oder benutzerdefinierten Typen.

Ich nenne diese prozessbegleitenden Komponenten *Mikrolithen*.[3] Ein Beispiel für dieses Refactoring sehen Sie in Folge 21 von [Shore 2020b]. Sie bieten die Aufteilung einer Komponente ohne operationale Komplexität.

3. Ich nenne sie Mikrolithen, weil ich sie mir ursprünglich als eine Kombination der besten Teile von Monolithen und Microservices vorgestellt habe. »Mikrolith« ist auch ein echtes Wort, das sich auf ein winziges Steinwerkzeug bezieht, das von einem größeren Stein abgeschlagen wurde, was fast als Metapher funktioniert.

ANMERKUNG

Meine Mikrolith-Refactorings sind die spekulativsten Refactorings. Ich habe sie nur an Spielzeugproblemen ausprobiert. Ich führe sie auf, weil sie einen Zwischenschritt zwischen Komponenten und Modulen darstellen.

Mikrolithen → Module

Mikrolithen sind stark isoliert. Sie sind praktisch Komponenten, die in einem einzigen Prozess laufen. Das führt zu einer gewissen Komplexität und einem gewissen Reibungsverlust.

Wenn Sie keine so starke Isolierung benötigen, können Sie die Top-Level-Schnittstellendatei und die Serialisierung/Deserialisierung entfernen. Rufen Sie einfach den Code des Mikrolithen normal auf. Das Ergebnis ist ein *Modul*. (Nicht zu verwechseln mit einer Quellcodedatei, die auch als Modul bezeichnet werden kann.)

Eine Komponente, die aus Modulen besteht, wird normalerweise als *modularer Monolith* bezeichnet, aber Module sind nicht nur für Monolithen geeignet. Sie können sie in jeder Komponente verwenden, egal wie groß oder klein sie ist.

Module → Neue Module

Wenn Ihre Module viele modulübergreifende Abhängigkeiten haben, können Sie sie möglicherweise vereinfachen, indem Sie ihre Zuständigkeiten neu ordnen. Dies ist eigentlich eine Frage der Anwendungsarchitektur und nicht der Systemarchitektur (mehr über die Entwicklung der Anwendungsarchitektur finden Sie im Abschnitt »Anwendungsarchitektur« auf Seite 555), aber ich erwähne es hier, weil es ein Zwischenschritt bei einer größeren Systemumgestaltung sein kann.

Big Ball of Mud → Module

Verwandte Themen

Inkrementeller Entwurf (S. 550),
Reflektierender Entwurf (S. 574)

Wenn Sie eine große Komponente haben, die sich in ein Chaos verwandelt hat, können Sie sie mithilfe vom evolutionären Entwurf schrittweise in Module umwandeln, wobei Sie die Daten nach und nach entwirren und aufteilen. Praful Todkar zeigt in [Todkar 2018] ein gutes Beispiel für diese Vorgehensweise. Auch dies ist eine Frage der Anwendungsarchitektur, nicht der Systemarchitektur.

Module → Mikrolithen

Wenn Sie eine starke Trennung wünschen oder eine große Komponente in mehrere kleine Komponenten aufteilen möchten, können Sie ein Modul in einen Mikrolithen umwandeln. Dazu führen Sie eine Top-Level-Schnittstellendatei ein und serialisieren komplexe Funktionsparameter.

Behandeln Sie den Mikrolithen so, als wäre er eine separate Komponente. Aufrufer sollten ihn nur über die Top-Level-Schnittstellendatei aufrufen und diese Aufrufe hinter einem Infrastruktur-Wrapper abstrahieren, wie im Abschnitt »Komponenten von Drittanbietern« auf Seite 567 beschrieben. Der Code des Mikrolithen sollte ähnlich isoliert sein; abgesehen von den allgemeinen Typen und Hilfsprogrammen, die eine Komponente verwenden könnte, sollte er nur andere Komponenten und Mikrolithen referenzieren, und zwar nur über deren Top-Level-Schnittstellen. Möglicherweise müssen Sie Ihr Modul zunächst umstrukturieren, um es komponentenähnlicher zu gestalten.

Netzwerkaufrufe sind viel langsamer und weniger zuverlässig als Funktions- und Methodenaufrufe. Die Umwandlung eines Moduls in einen Mikrolithen garantiert nicht, dass Ihr Mikrolith als vernetzte Komponente gut funktioniert. Theoretisch könnten Sie dafür sorgen, dass Ihre Mikrolithen als echte vernetzte Komponenten funktionieren, indem Sie eine Verzögerung von 1–2 ms in Ihre Top-Level-API einführen oder sogar zufällige Ausfälle. In der Praxis klingt das lächerlich und ich muss es erst selbst noch ausprobieren.

Mikrolithen → Komponenten

Wenn ein Mikrolith sich für die Verwendung als Netzwerkkomponente eignet, ist die Konvertierung in eine Komponente recht einfach. Es geht darum, die Top-Level-API-Datei in einen Server zu konvertieren und die Aufrufer so umzustellen, dass sie Netzwerkaufrufe verwenden. Dies ist am einfachsten, wenn Sie daran denken, ihre Aufrufe hinter einem Infrastruktur-Wrapper zu isolieren.

Die Umwandlung eines Mikrolithen in eine Komponente erfordert wahrscheinlich die Einführung von Fehlerbehandlung, Timeouts, Wiederholungsversuchen, exponentiellem Backoff[4] und Gegendruck (engl. Backpressure)[5], zusätzlich zu den betrieblichen und infrastrukturellen Änderungen, die die neue Komponente benötigt. Das ist viel Arbeit, aber das sind die Kosten der Vernetzung.

Module → Komponenten

Anstatt Mikrolithen zu verwenden, können Sie direkt von einem Modul zu einer Komponente wechseln. Obwohl dies durch Extrahieren des Codes möglich ist, sehe ich oft Leute, die stattdessen Module neu schreiben. Dies ist eine gängige Strategie beim Refactoring eines großen Big Ball of Mud, da der Code der Module oft nicht erhaltenswert ist. [Todkar 2018] demonstriert diesen Ansatz.

4. Anm. d. Übers.: »Exponentielles Backoff« bedeutet, dass der Kunde bei sich wiederholenden Fehlermeldungen des Anbieters längere Wartezeiten bei seinen Wiederholungsversuchen einhält.
5. Anm. d. Übers.: »Gegendruck« bezeichnet den Zustand, wenn ein Programm schneller Daten erhält, als es diese als Ergebnis bereitstellen kann – beispielsweise wenn ein Programm schneller aus einer Eingangsdatei lesen kann als das Ergebnis in die Ausgangsdatei zu schreiben.

Monorepo-Komponenten → Multirepo-Komponenten

Wenn Sie mehrere Komponenten im selben Repository haben, können Sie diese in separate Repositories extrahieren. Ein Grund dafür ist, dass Sie die Verantwortung für eine Komponente auf ein anderes Team übertragen. Möglicherweise müssen Sie gemeinsame Typen und Hilfsprogramme duplizieren.

Zusammengesetzte Refactorings

In der Regel werden Sie diese Refactorings auf Systemebene aneinanderreihen. Der gängigste Ansatz, den ich kenne, ist zum Beispiel die Bereinigung von Legacy-Code mit »Big Ball of Mud → Module« und »Module → Komponenten«. Oder, noch kompakter: Big Ball of Mud → Module → Komponenten.

Das Kombinieren von Komponenten ist ein ähnlicher Vorgang in umgekehrter Form: Multirepo-Komponenten → Monorepo-Komponenten → Mikrolithen → Module.

Wenn Sie eine Reihe von Komponenten mit verworrenen Zuständigkeiten haben, können Sie sie vielleicht zu neuen Zuständigkeiten mithilfe von Refactorings umbauen, anstatt sie neu zu schreiben: Komponenten → Mikrolithen → Module → Neue Module → Mikrolithen → Komponenten.

Voraussetzungen

Verwandtes Thema

Beseitigung von Hindernissen (S. 425)

Sie können die Ideen in dieser Praktik vermutlich nur mit den Komponenten verwenden, die Ihrem Team gehören. Auf Architekturstandards und Komponenten, die anderen Teams gehören, haben Sie wahrscheinlich keinen direkten Einfluss. Aber vielleicht können Sie die Leute beeinflussen, damit sie die gewünschten Änderungen vornehmen.

Änderungen an der Systemarchitektur hängen von einer engen Beziehung zwischen Entwicklerinnen und Betriebspersonal ab. Sie müssen zusammenarbeiten, um eine einfache Architektur für die aktuellen Anforderungen des Systems zu finden, einschließlich der Lastspitzen und der prognostizierten Steigerungen. Darüber hinaus müssen Sie sich weiterhin koordinieren, wenn sich die Anforderungen ändern.

Indikatoren

Wenn Sie Ihre Systemarchitektur gut weiterentwickeln:

- haben kleine Systeme kleine Architekturen und große Systeme verwaltbare Architekturen;

- ist die Systemarchitektur leicht zu erklären und zu verstehen;
- wird die unbeabsichtigte Komplexität auf ein Minimum reduziert.

Alternativen und Experimente

Verwandte Themen

Einfacher Entwurf (S. 563),
Inkrementeller Entwurf (S. 550),
Reflektierender Entwurf (S. 574)

Viele der Dinge, die wir als »evolutionäre Systemarchitektur« bezeichnen, sind eigentlich ganz normale evolutionäre Entwürfe. So ist beispielsweise die Migration einer Komponente von einer Datenbank zu einer anderen ein evolutionäres Entwurfsproblem, da es dabei hauptsächlich um den Entwurf einer einzelnen Komponente geht. Das Gleiche gilt für die Migration einer Komponente von einem Dienst eines Drittanbieters zu einem anderen. Diese Art von Änderungen werden von den evolutionären Entwurfsverfahren abgedeckt: einfacher Entwurf, inkrementeller Entwurf und reflektierender Entwurf.

Eine sich entwickelnde *System*architektur bedeutet, dass wir bewusst mit einem möglichst einfachen System beginnen und es entsprechend den sich ändernden Anforderungen erweitern. Das ist eine Idee, die noch nicht vollständig erforscht ist. Suchen Sie sich die Teile dieser Praktik aus, die für Ihre Situation geeignet sind, und sehen Sie dann, wie weit Sie damit kommen. Das eigentliche Ziel besteht darin, die Reibung zwischen Entwicklerinnen und Betriebspersonal zu verringern und die Fehlerbehebung zu erleichtern, ohne dabei die Zuverlässigkeit und Wartungsfreundlichkeit zu beeinträchtigen.

Weiterführende Literatur

Building Evolutionary Architectures [Ford et al. 2017] geht viel detaillierter auf architektonische Optionen ein. Die Autoren betrachten die Architektur aus der Sicht des Architekten und nicht aus der Sicht des Teams, wie ich es hier getan habe.

Building Microservices [Newman 2021] bietet einen klaren und gut geschriebenen Blick auf die Entwurfsprobleme und Kompromisse, die mit einer Microservice-Architektur verbunden sind, von denen viele auf die Systemarchitektur im Allgemeinen zutreffen.