SONDERDRUCK AUS
GERMAN TESTING MAGAZIN
Das unabhängige Magazin zu Software-Qualität

»Kompatibilitätstests mit CDC«

Consumer-Driven Contracts (CDC) sind ein Weg, um die Kompatibilität der Servicekommunikation sicherzustellen. Dieses Vorgehen gibt den Kommunizierenden ein Mittel an die Hand, welches ihre Interaktion mittels automatisierter Contract-Tests validiert. Aus welcher Motivation heraus ist dieses Konzept entstanden, wie funktioniert es grundlegend und was für Vor- und Nachteile ergeben sich aus dem Einsatz von CDC?

Der Trend hin zu verteilten Services stellt die Softwareentwicklung vor neue Herausforderungen. Insbesondere durch die Verwendung des Architekturmusters der Microservices ist die Geschäftslogik heutzutage auf mehrere Dienste verteilt. Eines der verbreitetsten Muster für den Informationsaustausch zwischen Microservices ist die Bereitstellung von Programmschnittstellen (APIs), welche wiederum häufig HTTP-basiert sind. An einer solchen Interaktion sind zwei Rollen beteiligt:

- der Dienstanbieter, genannt Provider, und
- der Aufrufer, genannt Consumer.

Motivation

Unabhängig von den eingesetzten Patterns und Technologien steigt die Komplexität mit der Anzahl an APIs und Consumer stetig an. Verfügen die Entwicklerteams nicht über das richtige Werkzeug, um ihre APIs bei Änderungen kontinuierlich zu überprüfen, erhöht sich das Risiko, Fehler zu übersehen.

Die größten Probleme ergeben sich aus der Ungewissheit über die Auswirkungen eines Release sowie aus hohem Aufwand durch wachsendes Risiko und gestiegenem Kommunikationsaufwand zwischen Teams. Dadurch entstehen häufig Symptome, wie unter anderem das Zunichtemachen von Funktionalitäten anderer Services (Breaking Changes), Verlangsamung der Auslieferung neuer Funktionen, Einschränkung bei der Evolution eines Service und in der Folge Unzufriedenheit der Kunden.

Einer der Ansätze, um Änderungen einer Programmschnittstelle zu verwalten, ist die Einhaltung des „Robustheitsgrundsatzes“ von Jon Postel [Wiki]:

- *Be conservative in what you do,*
- *be liberal in what you accept from others.*

Postel definiert damit einen Ansatz für die Evolution einer Programmschnittstelle. Der erste Teil konzentriert sich auf den Provider und bedeutet, dass er neue Attribute und Endpunkte frei hinzufügen, bestehende aber nicht entfernen oder umbenennen darf.

Der zweite Teil richtet sich an die Consumer und definiert, dass sie beim Konsumieren der Programmschnittstelle flexibel sein sollen und gegebenenfalls überschüssige Daten einfach ignorieren.

Folgt die Entwicklung diesem Robustheitsgrundsatz, sollte sie theoretisch nie in der Lage sein, Breaking Changes einzuführen. In der Praxis möchten Provider aber bestehende Felder verwerfen, entfernen oder aktualisieren. Dafür müssen sie die Frage *Wer verwendet das API wie?* beantworten können. CDC [Fow06] und deren Tests sind eine Antwort auf diese Frage.

Was ist ein Contract?

Der Contract ist eine Vereinbarung zwischen Consumer und Provider. Er definiert das beidseitige Verständnis über die Form ihrer

```
{
  "provider": { "name": "filmdatenservice" },
  "consumer": { "name": "filmbibiliothek" },
  "interactions": [
    {
      "description": "Hole Filme nach valider Bewertung",
      "request": {                                          //Beschreibt Aufruf
        "method": "GET",
        "path": "/filme",
        "query": "bewertung=8.0"
      },
      "response": {                                         //Beschreibt Antwort
        "status": 200,
        "headers": {
          "Content-Type": "application/hal+json"
        },
        "body": {
          "filme": [
            {
              "titel": "Iron Man",
              "imdbBewertung": 8
            }
          ]
        },
        "matchingRules": {                                  //Regeln für Inhalte
          "$.body.filme": {
            "match": "typ",
            "min": 0
          },
          "$.body.filme[*].titel": {
            "match": "typ"
          },
          "$.body.filme[*].imdbBewertung": {
            "match": "number"
          }
        }
}}]}
```

Listing 1: Beispiel-Contract in Form des Werkzeugs Pact [Pact]

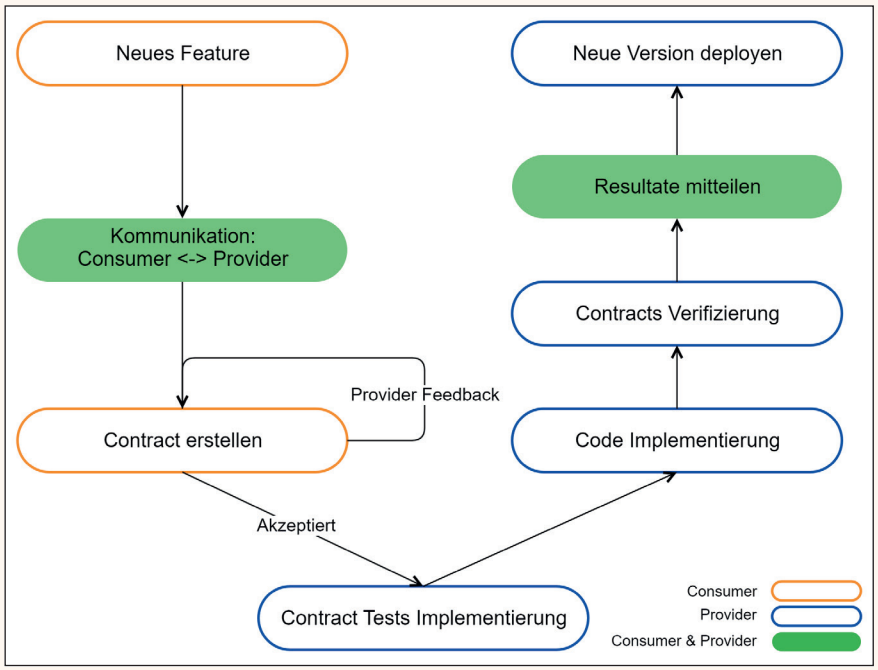

Abb. 1: CDC-Ablauf bei neuen Schnittstellen

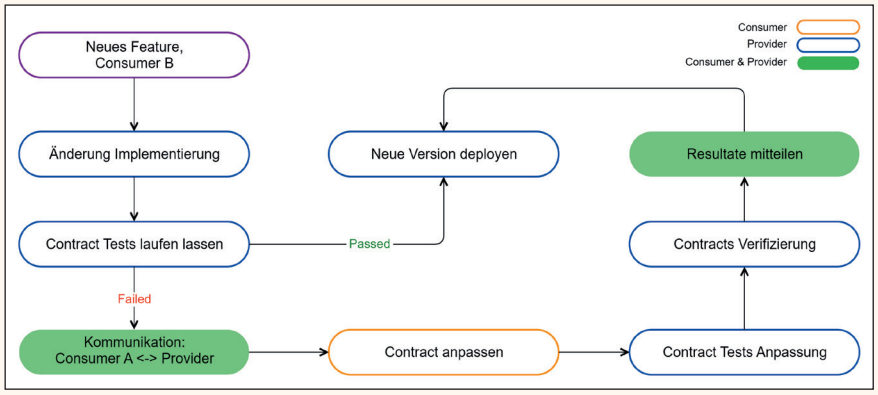

Abb. 2: CDC-Ablauf bei existierenden Schnittstellen

Kommunikation. Dies beinhaltet klare Regeln zu Art und Inhalt von Aufruf und Antwort. Beispielsweise enthält ein Contract für eine HTTP-basierte API-Interaktion ein Request/Response-Paar. **Listing 1** zeigt einen Beispiel-Contract.

Einordnung von Contract-Tests in die Testpyramide

In klassischen Integrationstests benötigt der Consumer einen realen Provider. Immer einen echten Service aufzurufen, macht die Tests allerdings fragil. Servicesimulation schaffte hier Abhilfe, dabei testeten Consumer gegen simulierte Provider. Doch laufen Entwickler schnell in die Situation, dass das simulierte Verhalten veraltet und der Test fälschlicherweise erfolgreich ist. Daher benötigt der Consumer ein Werkzeug, welches die Korrektheit der Simulation gegen den echten Provider validiert.

Contract-Tests adressieren diese Problemstellung. Die Tests laufen gegen das im Contract spezifizierte Verhalten des Providers. Parallel dazu wird der Contract auf der Seite des echten Providers validiert. Anhand des Contracts entsteht so der besondere Vorteil, dass beide Seiten isoliert voneinander ihre Integrationsfähigkeit prüfen können. Das macht Contract-Tests im Prinzip zu Integrationstests, welche aber von den Vorteilen der Servicesimulation profitieren.

Kommunizieren Consumer und Provider kaum oder gar nicht, zum Beispiel bei der Verwendung öffentlicher APIs, können Contract-Tests allerdings nicht sinnvoll verwendet werden.

Vorgehen bei CDC

Das Vorgehen bei CDC hat zwei mögliche Abläufe. Im ersten Ablauf wird ein neues Feature implementiert. Der zweite Ablauf zeigt eine notwendige Änderung der bestehenden Programmierschnittstelle. Bei CDC wird dies aus Sicht des Consumers definiert.

Abbildung 1 zeigt den Ablauf, wenn der Consumer eine neu zu implementierende Funktion benötigt. Beide Parteien setzen sich zusammen, um die API-Definition zu klären. Danach definiert der Consumer den Contract.

Der fertige Contract wird mit dem Provider geteilt. Dieser kann in dieser Phase solange Feedback geben („verhandeln"), bis er sich bereit erklärt hat, den Contract umzusetzen. Haben sich beide Seiten geeinigt, wird mit den Contract-Tests begonnen. Zu Beginn schlagen sie fehl, bis die Implementierung abgeschlossen ist. Die Umsetzung des Providers ist dann abgeschlossen, wenn alle seine Contract-Tests erfolgreich sind. Üblicherweise erhält der Consumer im nächsten Schritt die Ergebnisse der Verifizierung und kann sich auf das Release vorbereiten. Am Ende veröffentlichen Provider und Consumer die neuen Funktionen.

Die Abfolge in **Abbildung 2** beginnt, wenn eine Änderung der bestehenden Programmierschnittstelle erforderlich ist. Für ein leichteres Verständnis wird das Szenario aus der ersten Abbildung um einen zweiten Consumer B erweitert und der erste Consumer mit A gekennzeichnet. Stellt Consumer B neue Anforderungen an den Provider, durchläuft er die Ablaufschritte, die wir bereits gesehen haben.

Nachdem der Provider die neue Funktionalität implementiert hat, führt er wieder alle Contract-Tests durch. In unserem Fall sind das die Verträge von Consumer A und Consumer B. Bestenfalls sind die Ergebnisse positiv und der Provider integriert die neuen Änderungen direkt. Möglicherweise hat er jedoch Breaking Changes eingeführt und sieht diese anhand von fehlgeschlagenen Contract-Tests. Einen funktionierenden Stand erreicht der Provider nun über erneuten Austausch mit einem der Consumer.

Referenzen

- [Ata17] A. Atanasova, Introduction to Microservices Testing and Consumer-Driven Contract Testing with PACT, Novatec, 8.5.2017, siehe: https://www.novatec-gmbh.de/en/blog/introduction-microservices-testing-consumer-driven-contract-testing-pact/
- [Fow06] I. Robinson, Consumer-Driven Contracts, martinFowler.com, 12.6.2006, siehe: https://www.martinfowler.com/articles/consumerDrivenContracts.html#Consumer-drivenContracts
- [Pact] Pact, siehe: https://docs.pact.io/
- [Spring] Spring Cloud Contract, siehe: https://spring.io/projects/spring-cloud-contract
- [Wiki] https://de.wikipedia.org/wiki/Robustheitsgrundsatz#cite_note-1

Es gilt immer, den Lösungsweg mit der größten Nachhaltigkeit anzustreben. Der betreffende Consumer (im Beispiel A) nimmt im Folgenden die nötigen Anpassungen vor. Die Aktualisierung des Contracts folgt wieder dem Ablauf von **Abbildung 1**. Sobald der Provider die aktualisierte Fassung erhalten und umgesetzt hat, führt er erneut die Contract-Tests durch, um die Anpassungen zu validieren.

Schlussendlich kann die neue Version der Anwendung deployed werden und genießt die Sicherheit, dass es keine Breaking Changes in der Produktion gibt. Für den Fall, dass Consumer A einer Änderung des Contracts nicht zustimmt, sollten weitere Optionen und Gespräche geplant werden.

Das zuletzt beschriebene Vorgehen verwendet nur einen Provider und zwei Consumer, um das Basisverständnis der Wechselwirkungen zwischen den Parteien zu zeigen. In realen Projekten sind es sehr oft mehr als zwei Consumer. Daher empfiehlt es sich, große Teile des Prozesses durch Frameworks für Contract-Testen zu automatisieren. Besonders hilfreich sind Werkzeuge, die das Teilen und automatisierte Validieren anhand der Contracts unterstützen sowie die Verifikationsergebnisse konsolidiert anzeigen. Was nicht automatisiert werden darf, ist die Kommunikation zwischen Consumer- und Provider-Teams. Die Verhandlungen sind essenzieller Bestandteil für das Vermeiden von Missverständnissen und Fehlern. Als Beispiele für Werkzeuge seien an dieser Stelle Pact [Pact, Ata17] und SpringCC [Spring] genannt.

Organisatorische Vorteile und Herausforderungen

Bis hierher wurden die Auswirkungen des Einsatzes von CDC für den Entwicklungsprozess betrachtet. Die damit einhergehenden Herausforderungen lassen sich durch diverse Vorteile rechtfertigen.

Vorteile

Für Consumer ergeben sich zwei wesentliche Vorteile. Während des Test- und Feedbackprozesses ist es dem Consumer möglich, Fehler frühzeitig zu erkennen. Das minimiert Risiken für das Auftreten von Fehlern in Produktion, hervorgerufen von unabgestimmten Änderungen des Providers. Außerdem schärfen die kollaborativen Kommunikationszyklen das Verständnis der Provider-Schnittstelle aufseiten des Consumers. Dasselbe ist auch als Gewinn für den Provider zu sehen.

Einer der Hauptvorteile für den Provider besteht darin, dass er in der Lage ist, bereits in der Implementierungsphase versehentlich eingebaute Breaking Changes seines API zu entdecken. Darüber hinaus erhält er Transparenz über Nutzung der Programmierschnittstelle und kann ungenutzte Felder und Endpunkte bei Bedarf entfernen. Dadurch spart er sich den Aufwand, auf Verdacht zu implementieren, und kann sich auf wertschöpfende Funktionalitäten konzentrieren. Durch den umgekehrten Prozess liefern die Consumer nur tatsächlich genutzte Anforderungen an die Programmierschnittstelle.

Als Gewinn gegenüber Tests mit Servicesimulation erhalten automatisierte Contract-Tests eine garantierte Korrektheit der Gegenstelle. Der konstante Informationsaustausch über die Ergebnisse dieser Integrationstests lässt den Abstimmungsaufwand sinken und gleichzeitig Releasezyklen verkürzen. Diese Form der Tests stellt einen signifikanten Baustein für unabhängige, kontinuierliche Deployments dar.

Herausforderungen

CDC funktionieren nur schwer über Unternehmensgrenzen hinweg. Üblicherweise findet sich in öffentlichen APIs, wie von Facebook, kein Partner für Contracts. Die enge Kommunikation zwischen Provider und Consumer ist die Hauptvoraussetzung, um CDC zu implementieren. Auch global verteilte Teams, besonders in verschiedenen Zeitzonen, haben Herausforderungen bezüglich der Abstimmung. Nicht zu vergessen sind die nötigen Fähigkeiten zum Erstellen und Implementieren von CDC, die einen initialen Aufwand erfordern. Damit steigt die Hürde, um die Methode im Unternehmen etablieren zu können.

Fazit

CDC und deren automatisierte Tests bilden ein stabiles Vorgehen für die Entwicklung von APIs. Beide Kommunikationspartner profitieren vom Einsatz. Natürlich bedeutet die Einführung der Methode einen initialen Aufwand. Dieser rechnet sich allerdings in den meisten Fällen, abseits monolithischer Systeme, durch Risikominimierung, erhöhte Validierungsgeschwindigkeit und mehr Transparenz in der API-Entwicklung sehr schnell.

Antoniya Atanasova

antoniya.atanasova@novatec-gmbh.de
ist Consultant und Agile Quality Engineer bei der Novatec Consulting GmbH mit mehr als fünf Jahren Erfahrung in Agiler Entwicklung für verschiedene Kundenprojekte. Ihre aktuellen Schwerpunkte liegen in den Bereichen Microservices Testing und Entwicklung und Consumer-Driven Contracts.

Sebastian Letzel

sebastian.letzel@novatec-gmbh.de
ist Consultant und Agile Quality Engineer bei der Novatec Consulting GmbH. Seine aktuellen Schwerpunkte liegen in den Bereichen Testautomatisierung und Agile Softwareentwicklung.