Kontinuierliches API-Design und -Entwicklung

Überlegungen zum traditionellen Software-Lebenszyklus

von Daniel Kocot

Der altbekannte kontinuierliche Entwicklungslebenszyklus, dem man eigentlich in jedem Softwareentwicklungsprojekt begegnen sollte, stellt die Grundlage für die kommenden Ausführungen und Überlegungen zum Design von APIs dar.

Wir alle kennen den Entwicklungslebenszyklus aus **Abb. 1**.

Zusätzlich soll eben dieser Prozess durch die Einführung einer API-Management-Plattform unterstützt werden. Eine solche Plattform besteht in der Regel aus den folgenden Komponenten:

- Gateway
- Management-API/-UI
- Developer-Portal

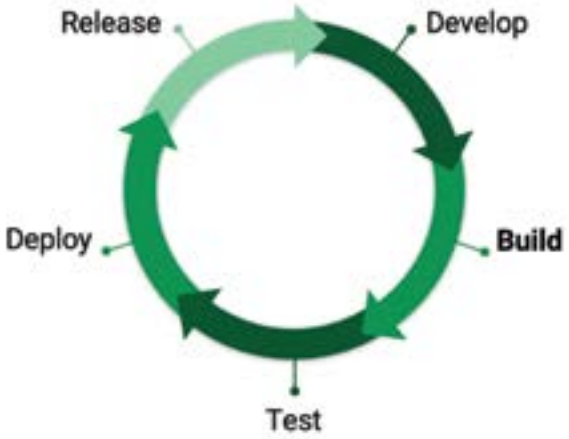

Abb. 1: Klassischer Entwicklungslebenszyklus

Gateway

Wenn man nach einer Definition für ein Gateway sucht, stellt man fest, dass das Gateway-Pattern eine gewisse Ähnlichkeit zum Pattern der Fascade aufweist. Laut der „Gang of Four" ist eine Fascade „ein Objekt, das als nach vorne gerichtete Schnittstelle dient und komplexeren zugrundeliegenden oder strukturellen Code maskiert" [1]. Chris Richardson beschreibt in seinem Buch „Microservices Patterns" das Gateway als Single-Entry-Point für alle Clients [2].

Management-API/-UI

Über das Management-API bzw. das -UI lässt sich die Plattform sowohl manuell als auch automatisiert konfigurieren.

Developer-Portal

Das Developer-Portal stellt den Zugriffspunkt für die Konsumenten (die Developer) dar. Über das Portal werden diese in die Lage versetzt, die angebotenen APIs für ihre jeweiligen Anforderungen zu nutzen. Die einfachste Form stellt hiermit die Darstellung als sogenannter „Catalog" dar. Durch Verwendung einer API-Management-Plattform innerhalb des Design- und Entwicklungsprozesses von APIs ergeben sich zwei unterschiedliche Anwendungsgruppen, die Produzenten und Konsumenten von APIs. Somit entstehen auch zwei unterschiedliche kontinuierliche Lebenszyklen. Auf der einen Seite betrachten wir einen Prozess der Veröffentlichung und des Managements, auf der anderen Seite den Konsumenten. Im Folgenden werde ich beide Prozesse näher beschreiben.

Abbildung 2 und 3 zeigen jeweils den Zyklus „API Publishing & Management“ sowie „API Developer/Consumer“

Nach der schematischen Darstellung der beiden Lifecycles innerhalb des Entwicklungsprozess soll nun der Blick auf die einzelnen Phasen gerichtet werden.
Als erstes wollen wir die Design-Phase betrachten, diese ist Teil des Lifecycles der Publishing- & Management-Sicht. Als Ersteller und API-Produzent sollten wir das API, das wir „nach außen“ exponieren wollen, wie ein Produkt betrachten. Um genau dem „API as a Product"-Ansatz zu folgen, müssen aber ein paar Dinge beachtet werden.

Öffentliche APIs

APIs werden grundsätzlich so entwickelt, als wären diese öffentlich zugänglich, also Public APIs. Durch diese Annahme ist man schon zu Anfang der Entwicklung gezwungen, darauf zu achten, dass die APIs für jeden potentiellen Konsumenten durch gute Konzeption, Implementierung und Dokumentation einfach zu verstehen und anzuwenden sind.

Entwicklung

Entwicklung ist grundsätzlich ein Prozess, der Kosten verursacht und daher ein Investment darstellt. Dieses Umstands sollte man sich bei der Betrachtung von APIs im Sinne eines Produkts bewusst sein. Bevor es an die konkrete Umsetzung geht, wird ein MVP (Minimum Viable Product) für das API-Produkt definiert. Dazu gehört dann auch eine Roadmap, um zu garantieren, dass die Konsumenten genau das bekommen, was diese tatsächlich benötigen. Durch diese Limitierung innerhalb der Entwicklung verhindern wir die Umsetzung nicht benötigter Features, wie z. B. ungenutzter Endpoints. Potentielle Sicherheitslücken werden so reduziert.

Abb. 2: API Publishing & Management

Abb. 3: API Developer/Consumer

Unveränderlichkeit

Was man bei der Entwicklung eines API auf keinen Fall außer Acht lassen sollte, ist die Unveränderlichkeit eben dieser. Im Sinne des Produktgedanken bedeutet dies, dass entsprechende Änderungen abwärtskompatibel sein müssen, da sonst Investitionen in das Produkt nicht sinnvoll sind.

Messbarkeit

Nachdem nun ein API-Produkt entwickelt und dieses den Konsumenten zur Verfügung gestellt wurde, kommt schnell die Frage nach dem Wert des Produkts auf. Hier gilt es nun, die Nutzung des Produkts entsprechend zu messen bzw. messbar zu machen. So kann auch einem nicht-technischen Publikum der Geschäftswert eines API-Produkts näher gebracht werden. Für dieses Publikum ist aber die Anzahl der APIs innerhalb des Produkts nicht entscheidend.

Sicherheit

Für API-Produkte ist Sicherheit ein wichtiger Faktor. Sie sollten daher durch OAuth, API Key Verification, XML/JSON Threat Protection und allgemeine Sicherheitsmechanismen geschützt sein.

Nach den Überlegungen zu einem möglichen Produktgedanken gilt es nun, sich innerhalb der Design-Phase mit der Umsetzung der Anforderungen zu beschäftigen. Hier gibt es zwei Möglichkeiten des Vorgehens: API-first und Code-first. Beide Ansätze sollen im Folgenden näher betrachtet werden.

Bei API-first stellt die OpenAPI Specification die Grundlage des weiteren Handels im Rahmen des Entwicklungsprozesses eines API dar. Hierbei handelt es sich um ein Beschreibungsformat für REST APIs. Mit der Spezifikation (**Listing 1**) ergibt sich die Möglichkeit, das gesamte API eines Services technisch zu beschreiben. Dies soll auch dazu führen, dass das API sowohl für Menschen als auch Maschinen als lesbar wahrgenommen werden kann. Es bleibt aber festzuhalten, dass es sich immer noch um eine rein technische Beschreibung, also eher eine Definition eines API handelt.

Um dem Ganzen einen deskriptiven Charakter zu verleihen, geht man mittlerweile dazu über, dies mithilfe von Application-Level Profile Semantics (ALPS [3]) durchzuführen. Mit diesem beschreibenden Format ist sichergestellt, dass ein einheitliches Verständnis zum jeweiligen API vorliegt. Dadurch entsteht zudem keine Abhängigkeit im Hinblick auf eine Technologie, ein Format und einen bestimmten API-Stil.

Mit der verfassten Beschreibung bzw. Definition geht es nun daran, APIs zu testen und gegebenenfalls auch zu mocken. Das Mocken von APIs, die in der OpenAPI Spec verfasst wurden, wird mit APISprout [4] ein ziemlich leichtes Unterfangen. APISprout stellt auf der Kommandozeile einen Mock-Server bereit und lässt sich unter Verwendung von Testcontainers in eine Build-Pipeline einbauen. Aber die Möglichkeit, ein API lokal testbar zu machen, reicht im ersten Schritt für eine kontinuierliche Entwicklung aus.

Um nun das gemockte API testen zu können, wird ein sogenannter REST Client benötigt. Hier gibt es eine Vielzahl

Listing 1

```
openapi: 3.0.2
info:
  description: This is a specification for an employee
                                                  service
  version: 1.0.0
  title: cc ERP Employee API
  license:
    name: MIT
servers:
  - url: http://api.codecentric.de

paths:
  /employees:
    get:
      summary: List all employees
      operationId: getEmployees
      tags:
        - employee
      responses:
        '200':
          description: A paged array of employees
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/
                                      EmployeeResponse'
              example:
                - id: 314
                  firstName: Daniel
                  lastName: Kocot
                - id: 315
                  firstName: Nils
                  lastName: Geistmann
        '500':
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
    post:
      summary: Create an Employee
      operationId: createEmployee
      tags:
        - employee
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/
                                  EmployeeCreateRequest'
      responses:
        '201':
          description: Employee successfully created
        '400':
          description: Validation error
        '500':
          description: Unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
  /employees/{employeeId}:
    get:
      summary: Info for a specific employee
      operationId: showEmployeeById
      tags:
        - employee
      parameters:
```

```
        - name: employeeId
          in: path
          required: true
          description: The id of the employee to retrieve
          schema:
            type: integer
      responses:
        '200':
          description: Expected response to a valid
                                                  request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/
                                      EmployeeResponse'
              example:
                id: 314
                firstName: Daniel
                lastName: Kocot
        '500':
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'

components:
  schemas:
    EmployeeResponse:
      type: object
      required:
        - id
      properties:
        id:
          type: integer
          format: int64
        firstName:
          type: string
        lastName:
          type: string
    EmployeeCreateRequest:
      type: object
      properties:
        lastName:
          type: string
        firstName:
          type: string
    Error:
      type: object
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

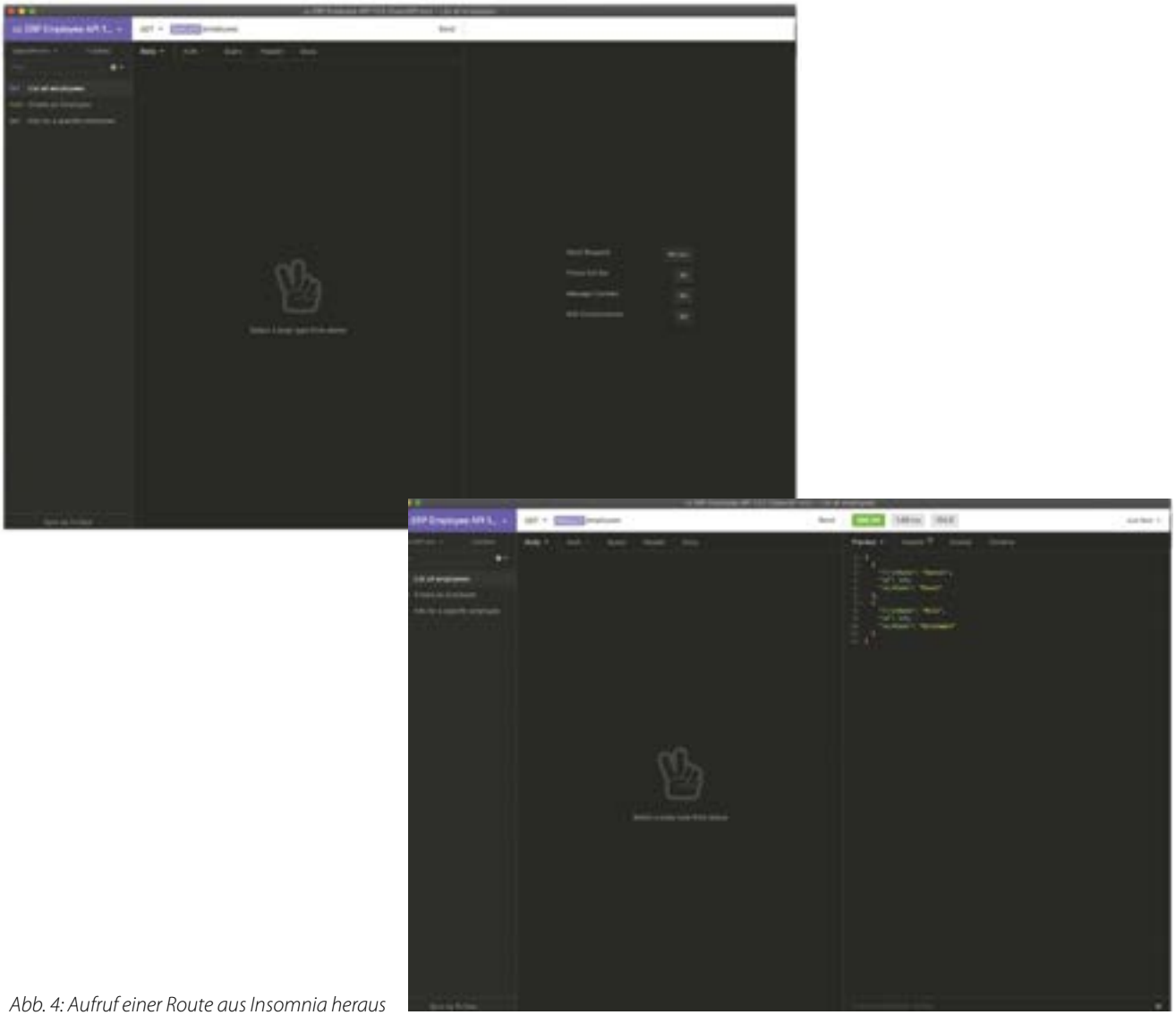

Abb. 4: Aufruf einer Route aus Insomnia heraus

entsprechender Tools. Für die Betrachtungen wird Insomnia [5] verwendet (siehe **Abb. 4**). Insomnia ist ein Open-Source-Produkt und wurde im Oktober 2019 von Kong übernommen. Mittlerweile wurde Insomnia in zwei Produkte aufgeteilt, Insomnia Core und Insomnia Designer. Zu beiden Produkten hat mein Kollege Pasquale Brunelli jeweils einen Blogpost geschrieben [6, 7]. Mit Insomnia können nun Requests auf die festgelegten API-Routen abgesetzt werden.

Sicherlich stellt sich jetzt der ein oder andere die Frage, ob API-first wirklich Vorteile bei der Entwicklung mit sich bringt. Bei der Entwicklung mit unabhängigen Teams bietet API-first hinsichtlich der Velocity der Teams einen entscheidenden Nutzen, da Design und Entwicklung nachweislich voneinander entkoppelt werden können. Natürlich ist es auch weiterhin möglich, dem Code-first-Ansatz zu folgen. Bei Code-first steht die Implementierung im Fokus. Hier sollte man nur schauen, dass die Entwicklung mit Bibliotheken wie Spring Boot REST Docs [8] oder auch FastAPI [9] unterstützt wird, um OpenAPI Specification direkt aus dem Service zu erzeugen.

Nun lässt sich die *Design*-Phase abschließen. Die API-Konsumenten befinden sich parallel hierzu in den *Discover-and-explore*-Phasen. Was bedeutet, dass sie dabei sind, die Welt des API-Produzenten zu betrachten und in Teilen auch zu entdecken. Zwingend dafür ist aber, dass auch APIs veröffentlicht wurden.

In der *Publish*-Phase wird das API über die Management-Plattform veröffentlicht. Hierbei kann die Implementierung des dahinterliegenden Services schon erfolgt sein, ist aber nicht zwingend erforderlich. Das Gateway als Teil der Plattform ist in der Lage, gestützt durch Plugins einen Mock des Services zu gewährleisten.
Die Automatisierung stellt in dieser Phase einen wichtigen Faktor dar, da nur durch sie auch ein zuverlässiger Veröffentlichungsprozess über alle Stages einer Entwicklungspipeline gewährleistet werden kann. Das API-Management-Portal steht dann auch in allen Stages zur Verfügung. Die Veröffentlichung erfolgt über das Developer-Portal.

Nachdem das API nun veröffentlicht wurde, kann der API-Konsument diese endlich testen und im besten Fall führt er sogar eine Subskription aus. Innerhalb des Portals steht das API nun zur Verfügung und wurde mit einer Dokumentation versehen. Das Portal bietet generell auch die Möglichkeit einer Sandbox zum Testen an, damit niemand direkt gegen ein API entwickeln muss. Für ein reales Testen gegen ein API muss auf der Konsumentenseite eine Buchung des gewünschten Service erfolgen.

Auf der Seite des Produzenten beginnt die spannendste Phase des API-Lifecycles. Seine API(s) wird bzw. werden benutzt, und nun gilt es den Betrieb der Plattform sicherzustellen. Und genau an dieser Stelle beginnt schon das Thema *Observabiltity* und zugleich die letzte Phase der beiden Zyklen.

Aber was verbirgt sich hinter dem Begriff *Observability*? Der Begriff Observability stammt ursprünglich aus der mathematischen Kontrolltheorie und ist eine Messgröße, die Rückschlüsse auf den inneren Zustand eines Systems zulassen soll. Geprägt wurde dieser Begriff von Robert E. Kalman. Dies kann auch auf Applikationen als Systeme übertragen werden. In unserem Fall wird die API-Management-Plattform die Rolle des zu beobachteten Systems einnehmen. Durch den Einsatz von Monitoring können Daten erfasst werden. Eine API-Management-Plattform bringt in der Regel einfaches Monitoring, aufgeteilt nach Services und Routen, mit sich. Sollte das Monitoring nicht hinreichend sein, kann auf bekannte Lösungen wie den ELK-Stack oder Prometheus in Kombination mit Grafana zurückgegriffen werden. Nun geht es in dem letzten Schritt darum, die aus dem Monitoring gewonnenen Daten zu analysieren. Dies kann manuell oder automatisiert erfolgen. Wichtig ist nur, dass diese Analyse wirklich stattfindet, um aus dieser entsprechend die nächsten Schritte abzuleiten.

Zusammenfassung

Kontinuierliche Prozesse sind ein wichtiger Bestandteil beim Design und der Entwicklung von APIs. Aufgrund der unterschiedlichen Beteiligten ergeben sich sogar zwei verschiedene Prozesszyklen. Unterstützt durch eine API-Management-Plattform, wird es relativ einfach, sich mit dem Thema zu befassen.

Seit Oktober 2016 ist ***Daniel Kocot*** *(daniel.kocot@codecentric.de) ein Teil des codecentric-Teams am Standort in Solingen. Schon seit Anfang der 2000er Jahre widmet er sich dem Thema der digitalen Transformation. Neben den aktuellen Schwerpunkten innerhalb von codecentric ist er auch Experte für den Einsatz von Produktinformationssystemen (PIM) und Database-Publishing mithilfe von Rendering-Technologien.*

Referenzen

[1] Erich Gamma, u. a. Design Patterns. Elements of Reusable Object-Oriented Software, 185ff.
[2] Chris Robertson, Microservices Patterns, 259
[3] http://alps.io/spec
[4] https://github.com/danielgtaylor/apisprout
[5] https://insomnia.rest
[6] https://blog.codecentric.de/2020/02/testen-und-debuggen-mit-insomnia/
[7] https://blog.codecentric.de/2020/06/einfuehrung-in-den-insomnia-designer/
[8] https://spring.io/projects/spring-restdocs
[9] https://fastapi.tiangolo.com