API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration

Gloria Bondel, Florian Matthes

TUM-I23101

API Management Pattern Catalog for Public, Partner, and Group Web APIs with a Focus on Collaboration

Gloria Bondel, Florian Matthes
Chair for Informatics 19
Software Engineering for Business Information Systems (sebis)
Technical University of Munich
Boltzmannstr. 3, 85748 Garching bei München, Germany
{gloria.bondel, matthes}@tum.de

December 2023
version 3.0

Acknowledgments

We want to thank all industry partners who participated in the creation of this Web API management pattern catalog for their valuable insights and feedback. Furthermore, we want to thank Andre Landgraf for the data collection and the design of the first pattern drafts.

Garching b. München, December 2023

Gloria Bondel
Prof. Dr. Florian Matthes

Abstract

Web Application Programming Interfaces (Web APIs) enable the emergence of platforms, efficient partner integration, reuse of functionality, or compliance (e.g., in banking), thus gaining attention from organizations. Managing Web APIs at the interface between several stakeholders inside and outside an organization makes API management an inherently collaborative organizational function. Nevertheless, explicitly formulated, context-dependent best practices for collaboration between stakeholders in API management are scarce.

Therefore, we present a pattern catalog comprising 22 patterns and 37 pattern candidates for managing Web APIs used across company borders, focusing on an API management team's interactions with external and internal stakeholders. The patterns apply to different types of organizations, including established organizations. Moreover, we relate the patterns to each other and to patterns belonging to previously published software pattern languages and catalogs. Also, the pattern catalog defines a standardized taxonomy used throughout the patterns.

Contents

Acknowledgments **i**

Abstract **ii**

1. Introduction **1**
- 1.1. Pattern Catalog Objective . . . 2
- 1.2. Pattern Catalog Design Approach . . . 3
  - 1.2.1. Data Collection and Analysis . . . 4
  - 1.2.2. Iterative Improvements . . . 5

2. API Management Pattern Catalog Taxonomy **8**
- 2.1. Software Artifacts in Web API Management . . . 8
- 2.2. Stakeholders in Web API Management . . . 10

3. Structure of the API Management Pattern Catalog **12**
- 3.1. Elements of the API Management Pattern Catalog . . . 12
- 3.2. Visualization of the Pattern Catalog Structure . . . 15
- 3.3. API Management Pattern Summaries . . . 16
  - 3.3.1. Interface Type Patterns Summaries . . . 16
  - 3.3.2. API Provider Internal Patterns Summaries . . . 17
  - 3.3.3. API Consumer-facing Patterns Summaries . . . 18

4. Interface Type Patterns **19**
- 4.1. Web API . . . 20
- 4.2. Client Library . . . 26
- 4.3. Frontend Venture . . . 34

5. API Provider Internal Patterns **39**
- 5.1. API-as-a-Product . . . 40

5.2. API Product Owner 49
5.3. Collaborative Pilot Project 54
5.4. Play-it-fast Approach 59
5.5. Idea Backlog 64
5.6. Testing Strategy 69
5.7. API Clearing Process 74
5.8. API Facade 79
5.9. API Quality Monitoring 85

6. API Consumer-facing Patterns 92
6.1. Role-based Marketing 93
6.2. Customer Success Stories 98
6.3. Newsletter 103
6.4. Consumer-centric API Description 108
6.5. Integration Guide 113
6.6. Onboarding Self-service 120
6.7. Integration Partner Program 128
6.8. API provider-wide ticketing management 134
6.9. Dedicated Support Team 139
6.10. Service-Level Agreement (SLA) 146

7. Pattern Candidates 153
7.1. Interface Type Pattern Candidates 153
7.2. API Provider Internal Pattern Candidates 154
7.3. API Consumer-facing Pattern Candidates 156

8. Summary 160

A. Expert Interviews 162

B. Case Base 163

C. Related patterns and pattern languages/catalogs 165

List of Figures 167

List of Tables 169

Bibliography 170

CHAPTER 1

Introduction

Web Application Programming Interfaces (Web APIs)[1] are the defacto leading approach to data exchange between organizations. A Web API is an interface between two software components that defines how these components interact (De, 2017; Jacobson et al., 2012) and uses HTTP(S) as communication protocol (Bermbach & Wittern, 2016; De, 2017). More precisely, the components leverage HTTP as data transport protocol, e.g., SOAP/WSDL, or as an application protocol that additionally dictates the semantics of the API behavior, e.g., RESTful APIs (Daigneau, 2011). Common architectural styles of Web APIs are *REST (REpresentational State Transfer)* (Fielding, 2000), *RPC-style (Remote Procedure Call)* (Maleshkova et al., 2010; Santoro et al., 2019), and *SOAP*[2] (Santoro et al., 2019; Spichale, 2017) architectures.

Nowadays, Web APIs are pervasive as interfaces enabling inter-organizational data exchange. An international survey questioning 800 IT leaders revealed that at the beginning of 2021, 96% of organizations used public or private APIs (MuleSoft, 2021). The organizations using APIs report, among others, increased productivity and more innovation (MuleSoft, 2021). Moreover, the public sector recognizes the increasing importance of Web APIs for the digital transformation of government services (Santoro et al., 2019). For example, in 2018, the European Commission launched the *"APIs4DGov: APIs for Digital Government"* study to analyze the use of Web APIs in governmental initiatives across Europe (Vaccari et al., 2020). The study found that Web APIs are valuable assets for different European government institutions.

Current Information Systems (IS) research conceptualizes Web APIs as resources at the interface between an organization and third-party developers (de Reuver et al., 2018; Eaton et al., 2015; Ghazawneh & Henfridsson, 2010, 2013; Karhu et al., 2018) that enable the emergence of platforms (de Reuver et al., 2018; Eaton et al., 2015; Ghazawneh & Henfridsson, 2010,

[1]In the remainder of this document the term *'API'* refers Web APIs if not explicitly stated otherwise.

[2]Since the publication of SOAP version 1.2, SOAP is not an acronym of *Simple Object Access Protocol* anymore (Gudgin et al., 2007).

2013; Karhu et al., 2018), the realization of new business models (De, 2017; Medjaoui et al., 2018), efficient partner integration (Jacobson et al., 2012), or compliance (e.g., in banking). Therefore, Web APIs are resources with strategic value (Yoo et al., 2010), and firms need to design and maintain them carefully. The organizational function responsible for designing and maintaining Web APIs as well as additional technical and social resources that enable consumers to use an API is commonly denoted as *API Management*.

Hence, the management of Web APIs increases in importance for organizations.

1.1. Pattern Catalog Objective

However, the analysis of current research and practice-driven API management literature reveals some gaps.

First, most current research on API management focuses on ex-post analyses' of successful platforms controlled by tech-giants like Apple[3] and Google[4] (de Reuver et al., 2018), neglecting the perspective of established organizations in traditional industry sectors. Moreover, the analyzed organizations are mostly located in entrepreneurial regions in North America, especially Silicon Valley. Also, the research lacks clear instructions on API management for practitioners.

More specific API management guidelines, including patterns, can be found in the practice-driven API management literature. Yet, most of the practice-driven literature is concerned with rather technical aspects of API management, e.g., how to achieve RESTful compliance or improve security using particular authentication and authorization mechanisms (De, 2017).

Finally, an API management function cannot exist in isolation, but ongoing collaboration and knowledge transfer (Islind et al., 2016) with various stakeholders inside and outside the organization is necessary. Nevertheless, to the best of the authors' knowledge, only a few best practices for knowledge transfer and collaboration in API management have been explicitly formalized.

As a result, the research objective of this pattern catalog is:

The identification of **API management patterns** focusing on **collaboration** in **public**, **partner**, and **group** API initiatives, for different types of API provider organizations, including **established organizations** located in **Europe**.

In the following, we break down the research objective into its aspects and elaborate on each of them:

- *Pattern form.* Web API management initiatives have different characteristics and aim to achieve different strategic goals. Therefore, API management best practices need to consider the context in which an organization wants to apply them. Thus, we

[3]https://www.apple.com/de/ (accessed 20.12.2023).
[4]https://www.google.com/ (accessed 20.12.2023).

choose to document Web API management best practices as patterns. The purpose of patterns is to enable knowledge dissemination in the respective domain (Buckl et al., 2013). In addition, patterns document operational knowledge on current practices for practitioners and encode knowledge on the evolution of a discipline for academia (Buckl et al., 2008; Buckl et al., 2013; Khosroshahi et al., 2015). Also, recording best practices as patterns allows for the stepwise creation and adaptation of an API management function tailored to the context of an organization (Khosroshahi et al., 2015). In software engineering, several pattern languages have already been successfully introduced, e.g., by Buschmann et al. (1996) and Gamma et al. (1994).

- *Collaboration.* We aim at identifying API management patterns focusing on collaboration between an API management team and other internal and external stakeholders of a Web API initiative.
- *Organizational affiliation.* The pattern catalog focuses on API initiatives with the API provider and API consumer belonging to different organizations, i.e., public, partner, and group API initiatives.
- *Organizational scope.* We derive the patterns by analyzing API initiatives of different types of organizations, including established European and young digital organizations seated in the US. Thus, the pattern catalog applies to different types of organizations, including established organizations in Europe.
- *API provider perspective.* While we aim to encode knowledge for all stakeholders involved in API management, we document them from the perspective of the API provider.

Additionally, we present consistent API management terminology, allowing for easy pattern integration. Also, standardized terminology fosters the communication and evolution of patterns (Khosroshahi et al., 2015).

The patterns and terminology contribute to practice since API management teams can explore proven solutions for specific concerns, considering their respective API initiative characteristics. Additionally, they can use the pattern catalog to benchmark their practices with proven solutions. The scientific contribution is the documentation of state-of-the-art collaborative activities between API management stakeholders that provides a basis for theorizing on collaboration and knowledge transfer within and outside an organization. In the future, we aim to extend and revise the API management pattern catalog based on new insights from existing or new cases.

1.2. Pattern Catalog Design Approach

The goal of the research approach is to create a pattern catalog supporting API provider teams in API management activities focusing on collaboration with stakeholders. Hence, we collected data from different data sources, analyzed the data, and iteratively improved the pattern catalog based on feedback from different stakeholders[5]. We will discuss the data collection and analysis, and the iterative improvements in the following.

1.2.1. Data Collection and Analysis

Each pattern description combines information from expert interviews, publicly available information on selected public API initiatives, other software pattern languages and catalogs, and practice-driven books on API management. In the following, we will describe each of these data sources.

Expert Interviews

Since API management patterns aim at capturing current best practices, we interviewed practitioners working in API management to collect data as previously reported in Bondel et al. (2022) and Landgraf (2021). We conducted 16 semi-structured interviews with 15 interviewees between August 2020 and January 2021. An overview of the interviews is provided in Appendix A. The interviews focused on past and current tasks or issues that the interviewees face in their API management daily work. Furthermore, we discussed solutions to address the tasks or solve the issues successfully. From these interviews, we created a case base holding 14 unique cases. We enriched the information from the interviews with publicly available information on the cases, if available. Moreover, we assigned each case an ID consisting of the letter 'C' and a number (see Appendix B). In the remainder of this catalog, we use the ID to link information on practices to the cases in which we observed them to enable reproducibility of results.

In parallel, we analyzed the collected data by applying a Grounded Theory Methodology (GTM) approach that guides qualitative content analysis in Information Systems (IS) research (Wiesche et al., 2017). We first identified pattern candidates. Next, we analyzed which pattern candidates can be conceptualized as genuine patterns applying the *rule of three* (Coplien, 1996) (also applied by Buckl et al. (2008), Khosroshahi et al. (2015), and Uludağ et al. (2019)). The rule of three states that *"[...] a good pattern should have three examples that show three insightfully different implementations."* (Coplien, 1996, p. 35).

The data analysis was conducted by two researchers, with the second researcher reviewing and building on the results of the initial data analysis[6].

[5]Formally we applied a design science research approach (Hevner, 2007; Hevner et al., 2004). However, for reasons of readability, we summarize the process into data collection, analysis, and iterative improvements.

[6]The results of the first data analysis are published in student thesis Landgraf (2021). The second researcher used the initially identified pattern and pattern candidate names as input to the second round of data analysis to validate and refine the results.

Selected Public API Initiatives

In addition, we add the two public API initiatives Stripe[7], and Twilio[8] as additional cases to the case base, i.e., we systematically reviewed if and how these public API initiatives apply previously identified pattern candidates. These two cases add further insights since they represent initiatives of organizations that use Web APIs as their primary product distribution channel. An overview of the final case base is presented in Appendix B.

Software Pattern Catalogs and Languages

Moreover, we analyzed existing software pattern catalogs and languages to identify related, alternative, or overlapping patterns and enrich our pattern descriptions. Tab. 1.1 presents an overview of the reviewed pattern collections. An overview of the relation between these software patterns and the Web API management patterns presented in the pattern catalog at hand is provided in Appendix C.

Practice-driven API Management Literature

Moreover, we systematically reviewed relevant, practice-driven literature on API management to enrich and validate our pattern descriptions. The reviewed practice-driven Web API management literature comprises De (2017), Jacobson et al. (2012), Medjaoui et al. (2018), and Spichale (2017).

1.2.2. Iterative Improvements

In addition to enriching the pattern descriptions with information from different sources, we iteratively improved the pattern catalog based on feedback from the scientific pattern community and practitioners. In the following, we will describe the feedback and adoption processes.

Feedback from the Scientific Pattern Community

We adopted the API Management Pattern Catalog to meet the requirements and best practices of the scientific software engineering pattern community. We collected feedback from the scientific pattern community by participating in the European Conference on Pattern Languages of Programs 2021 (EuroPLoP'21)[9]. The Hillside Group[10], the defacto leading research community for software engineering pattern creation, organizes the conference. Significant changes resulting from the feedback are a new pattern structure and a new approach to relating patterns. As part of this process, we published Bondel et al. (2022), which presents an intermediate summary of the pattern catalog and initial description of two patterns.

[7]https://stripe.com/ (accessed 20.12.2023).
[8]https://www.twilio.com/ (accessed 20.12.2023).
[9]https://www.europlop.net/content/conference (accessed 20.12.2023).
[10]https://hillside.net/ (accessed 20.12.2023).

Table 1.1.: Overview of pattern languages and catalogs reviewed and related to patterns in this API management pattern catalog.

Name of the Pattern Language/Catalog	**Source**
Patterns for API Design (previously known as *Microservice API Patterns (MAP)*)	Zimmermann et al. (2017), Stocker et al. (2018) and Zdun et al. (2018), Lübke et al. (2019), Zimmermann, Lübke, et al. (2020), Zimmermann, Pautasso, et al. (2020), Zimmermann et al. (2019), and Zimmermann et al. (2022)
Design Patterns	Geewax (2021)
Patterns for RESTful Conversations	Pautasso et al. (2016)
Control-Flow Patterns for Decentralized RESTful Service Composition	Bellido et al. (2013)
Service Design Patterns	Daigneau (2011)
SOA Design Patterns	Erl (2008)
SOA Patterns	Rotem-Gal-Oz (2012)
Microservices Patterns	Richardson (2019) and Richardson (n.d.)
Microservices Patterns	Newman (2019) and Newman (n.d.)
Enterprise Integration Patterns	Hohpe and Woolf (2003) and Hohpe (n.d.)
Remoting Patterns	Völter et al. (2004)
Patterns for Enterprise Application Architecture	Fowler (2003)
Object-oriented Software Design Patterns (also known as *Gang of Four* or *GoF* book)	Gamma et al. (1994)
Pattern-Oriented Software Architecture (also known as *POSA* series)	Buschmann et al. (1996), Schmidt et al. (2000), Kircher and Jain (2004), and Buschmann et al. (2007a, 2007b)
Patterns for High-Capability Internet-Based Systems	Dyson and Longshaw (2004)

Feedback from Practitioners

We used a survey to collect feedback on the pattern catalog's applicability, comprehensibility/usability, completeness, and correctness from a practitioner's viewpoint. Overall, 18 practitioners responded to the survey. The practitioners evaluated the pattern catalog as applicable to real-world settings, understandable, usable, and correct. Also, individual pattern descriptions meet the readers' information needs. However, future work should comprise the identification of further patterns and the evolution of existing patterns. Also, the pattern catalog should be published as an HTML document to improve its searchability and navigability.

CHAPTER 2

API Management Pattern Catalog Taxonomy

The basic concepts and relationships between software artifacts and stakeholders in API management are illustrated in Fig. 2.1. The model aims to create a clear and shared understanding and basic naming conventions for the pattern catalog. A consistent taxonomy is an essential aspect of a pattern language (Buckl et al., 2008; Buckl et al., 2013; Khosroshahi et al., 2015). The model is derived from literature and interview data and has previously been published in Bondel et al. (2022)[1]. We will first focus on the software artifacts, followed by the stakeholders involved in API management.

2.1. Software Artifacts in Web API Management

The basic concepts and relationships between software artifacts in API management are illustrated in the lower part of Fig. 2.1.

> "The API management software artifacts are a backend, an API gateway, a web API, an API developer portal, and the API-consuming [client] application. The *backend* can be any software component that provides a particular capability, i.e., functionality or data, that should be made accessible to other stakeholders. The *Web API* is used to make the functionality or data of the backend accessible. A Web API is an interface available over the public internet and thus can be accessed using the HTTP protocol. An *[client] application* integrates the functionality or data provided via the Web API. This application could itself also be a backend providing (enhanced) functionality or data to further applications.

[1]In this chapter, we highlight direct quotes of Bondel et al. (2022) using quotation marks, indenting the respective paragraphs, and adding the source at the end of the quote (i.e., – (Bondel et al., 2022)). A quote can span several paragraphs. We highlight changes using squared brackets.

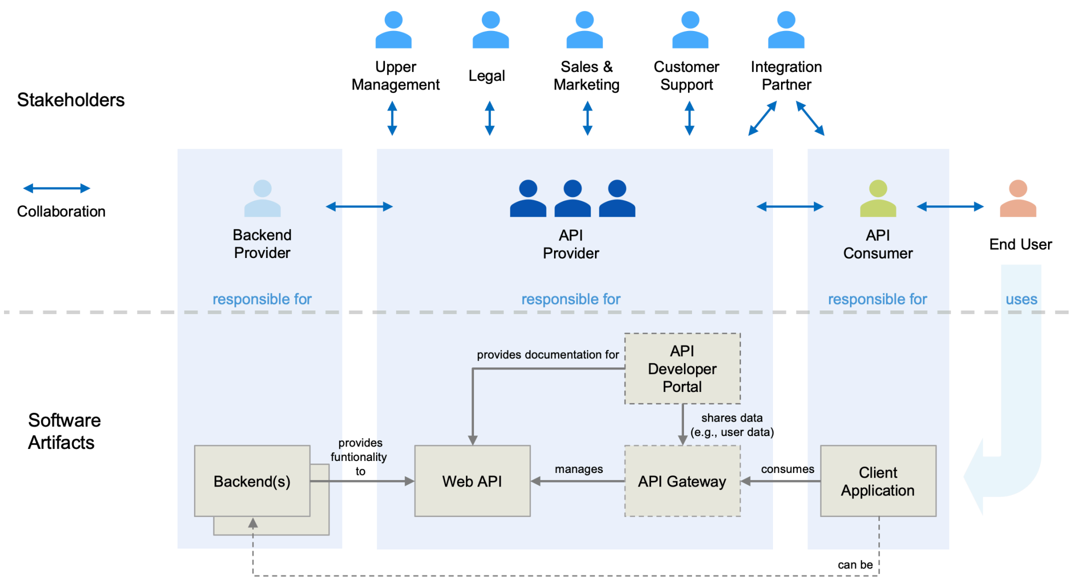

Figure 2.1.: Conceptual overview of software artifacts and stakeholders involved in API management (adapted from Bondel et al. (2022)).

Besides, two types of software platforms support API management, i.e., the API gateway and the API developer portal. The *API gateway* is a reverse proxy that intercepts incoming requests from clients. Hence, an API gateway is an infrastructure platform managing the API provider and consumer interaction at runtime (Zimmermann et al., 2022). Management capabilities provided by the API gateway include authentication, rate limiting, statistics and analytics, monitoring, policies, alerts, and security [(De, 2017)]. The *API developer portal* is a web application that allows the API provider to communicate information to API consumers needed to find and use an API. This information should comprise the API documentation including information on the APIs location (URI) and the structure of valid API calls. Additional information can contain, among other things, terms and conditions, contact information, marketing information, changelogs, prizing information, and social content like forums or blogs (De, 2017). Also, self-services supporting the API consumer can be part of the API portal, e.g., a self-service for API consumer registration (De, 2017). Often, the API portal shares data with the API gateway, for example to ensure that only registered users can make requests to an API. Both API management software components, the API gateway and the API developer portal, are optional.

An *API initiative* denotes any new or ongoing endeavor of providing a Web API to potential consumers, irrespective of the use of an API gateway or API portal."

– (Bondel et al., 2022)

2.2. Stakeholders in Web API Management

We identify nine stakeholder groups as illustrated in the upper part of Fig. 2.1.

"[...] We visualize the collaboration between the API provider and other stakeholders. While the other stakeholders also communicate, we did not illustrate those collaboration flows for conciseness reasons. Also, a stakeholder role does not have to be occupied by one or more dedicated persons. Instead, a person could have several roles.

The *API provider* is responsible for carrying out API management tasks. The API management comprises all tasks related to designing and maintaining the Web API, the API gateway, and the API developer portal. An API provider team includes business and technical roles, with business roles defining and pursuing business goals and technical roles aiming to ensure technical KPIs (Medjaoui et al., 2018). The tasks of an API provider comprises all activities aimed towards realizing the goals defined by the API management lifecycle. In many cases, backend developers also design and maintain the respective Web API and thus occupy two roles.

The API provider team also collaborates with the *API consumer*. The API consumer is the team that integrates the Web API into its application. Such an application can be either an end-user application or an application that other API consumers use, e.g., an API wrapper. Thus, communication can include inter alia passing on additional information on the API functionality, change requests, or issue reporting. The API consumer can belong to a different organization than the API provider team. The differences in the organizational affiliation between the API provider and the API consumer are discussed in the literature (De, 2017; Jacobson et al., 2012; Santoro et al., 2019). Accordingly, if the API provider and the API consumer belong to the same organization, the API is categorized as a private API. If the actors belong to different organizations, the API can be a partner or a public API. Partner APIs are accessible only for selected external partners, while public APIs are accessible to every interested developer. The presented API management pattern language focuses on partner and public APIs. The *end-user* denotes the person using the application and thus mainly communicates with the API consumer who provides the application.

The API provider also collaborates with the *backend provider* who designs and maintains the backend. Here, the collaboration mainly focuses on change requests and bug reporting for backend functionality that the API consumer or monitoring tools report to the API provider team. The API provider and the backend provider often belong to the same organization, thus the organization makes its own functionality or data accessible to external consumers. However, there are two settings in which the API provider and the backend provider belong to different organizations. In the first setting, the API provider operates a marketplace, that allows other backend and API providers to publish APIs. Still, all API marketplaces that are part of our case base also offer APIs provided by the API providers organization. Secondly, we observed two cases in corporate group settings where the API provider is employed by one subsidiary firm focused on IT provision, and the backend providers are other subsidiaries in the group. Nevertheless, the groups' overall goals still guide the interaction between the backend provider and the API provider.

Additionally, several other stakeholders can interact with the API provider team. First of all, the *upper management* can support the API provider, e.g., by promoting API management in strategic initiatives. The API management team needs to involve the *legal department* to ensure privacy conformance of APIs. Moreover, the legal department can negotiate contracts with partners using the APIs. *Sales & Marketing* supports the marketing activities for new APIs or enforces compliance with corporate identity specifications. [The *integration partner* supports API consumers lacking technical capabilities to integrate a Web API.] Finally, an existing *customer support* can be a contact point for API consumers, which then forwards tickets to the API provider team. All these stakeholders belong to the same organizations as the API provider.

Our pattern language focuses on collaboration patterns for the API provider."

– (Bondel et al., 2022)

This pattern catalog presents Web API management patterns from the API provider perspective.

CHAPTER 3

Structure of the API Management Pattern Catalog

In this section, we describe the elements of the API management pattern catalog, provide a visual overview of the instantiated patterns and their relationships, and present summaries of each pattern.

3.1. Elements of the API Management Pattern Catalog

This API management pattern catalog consists of patterns and their relations.

A *pattern* describes an abstract solution to a problem that captures the solution's core without prescribing any implementation details (Alexander et al., 1977; Henfridsson et al., 2014). Thus, API management teams can reuse patterns and adapt them to different contexts (Buckl et al., 2013). A pattern has to meet the *rule of three* (Coplien, 1996), i.e., a solution approach is only a pattern if we observed its successful implementation in at least three cases. Thus, the presented patterns are proven solutions for collaboration in API management.

A *pattern catalog* denotes a collection of related patterns that together do not entirely cover a problem domain (Coplien, 1996). Since the presented patterns do not completely cover the API management problem domain, the document at hand is a pattern catalog. In comparison, a *pattern language* is a collection of patterns and their relationships that address a problem domain exhaustively (Alexander et al., 1977; Meszaros & Doble, 1997). Many of the most renowned and used pattern collections are pattern catalogs (Coplien, 1996), e.g., Buschmann et al. (1996) and Gamma et al. (1994).

Patterns should have a consistent structure for convenience and clarity reasons (Alexander et al., 1977). Therefore, each pattern in this catalog has the same set of mandatory and optional elements. We describe each of these elements in detail in the following:

- Each pattern has a **name** and potentially one or more **aliases**. A name allows readers to find relevant patterns quickly and can become part of an API management team's vocabulary (Coplien, 1996).

- Furthermore, each pattern belongs to one of the **pattern categories** *Interface Type Pattern*, *API Provider Internal Patterns*, and *API Consumer-facing Patterns*. Interface Type Patterns capture different approaches to making functionality and data provided via Web API available to API consumers. API Provider Internal Patterns describe patterns that require the API provider team to collaborate mainly with API provider organization internal stakeholders. In contrast, API Consumer-facing Patterns are concerned with the interaction between the API provider and consumer.
- A short **summary** of each pattern allows the reader to grasp the essence of a pattern quickly.
- A **sketch** visualizes the patterns basic concept (Coplien, 1996).
- The **context** describes the situation in which the reader can apply a pattern (Meszaros & Doble, 1997). The situation imposes constraints that the solution needs to address (Meszaros & Doble, 1997).
- A **concern** captures the (design) problem that the pattern addresses (Coplien, 1996; Meszaros & Doble, 1997). Concerns usually represent the interests and goals of the stakeholders applying a pattern (Buckl et al., 2008; Khosroshahi et al., 2015; Uludağ et al., 2019), i.e., the API provider team.
- Patterns address concerns that are difficult to solve due to contradictory goals and considerations of stakeholders (Coplien, 1996; Meszaros & Doble, 1997). The **forces** describe these trade-offs (Coplien, 1996; Meszaros & Doble, 1997). The context usually indicates which forces the pattern should optimize (Meszaros & Doble, 1997). Moreover, understanding the forces allows the reader to better understand the concern and the solution (Coplien, 1996).
- The **solution** captures the core approach to solving the concern in a given context (Meszaros & Doble, 1997). The solution provides enough detail to enable the API provider team to apply the pattern. Nevertheless, simultaneously, the solution is generic enough to apply to many contexts (Coplien, 1996). Furthermore, the solution dictates how the forces are resolved (Meszaros & Doble, 1997). In some cases, we observed **variants** of a solution.
- The **stakeholders** list all roles involved, affected, or influenced by API management. Stakeholders can be internal or external to the API provider team's organization. This pattern catalog focuses on the API provider team, including the Web API provider, the API gateway provider, the API developer portal provider, and the API governance role.
- The **implementation hints** provide additional information supporting the successful implementation of the pattern.

- The **consequences** pick up on the forces and explain which considerations the solution optimizes at the expense of others (Gamma et al., 1994).
- The **related patterns within this pattern catalog** section relates a pattern to other patterns presented in this pattern catalog.
- Similarly, **other related patterns** point to patterns already published by other authors. Tab. 1.1 provides an overview of the reviewed pattern catalogs and languages. This section is only instantiated if the respective pattern relates to any of these other patterns.
- **Known uses** describe the cases in which we observed the pattern and, if possible, provide some details on its implementation. Since we apply the *rule of three* (Coplien, 1996), each pattern has at least three known uses.

Finally, we also present pattern candidates. Pattern candidates are solution approaches we observed in our case base but did not meet the *rule of three* (Coplien, 1996), i.e., we observed them only in one or two cases. However, they bear the potential to become patterns in the future. Hence, we briefly describe the essence of each pattern candidate.

3.2. Visualization of the Pattern Catalog Structure

In Fig 3.1, we make the relations between patterns explicit.

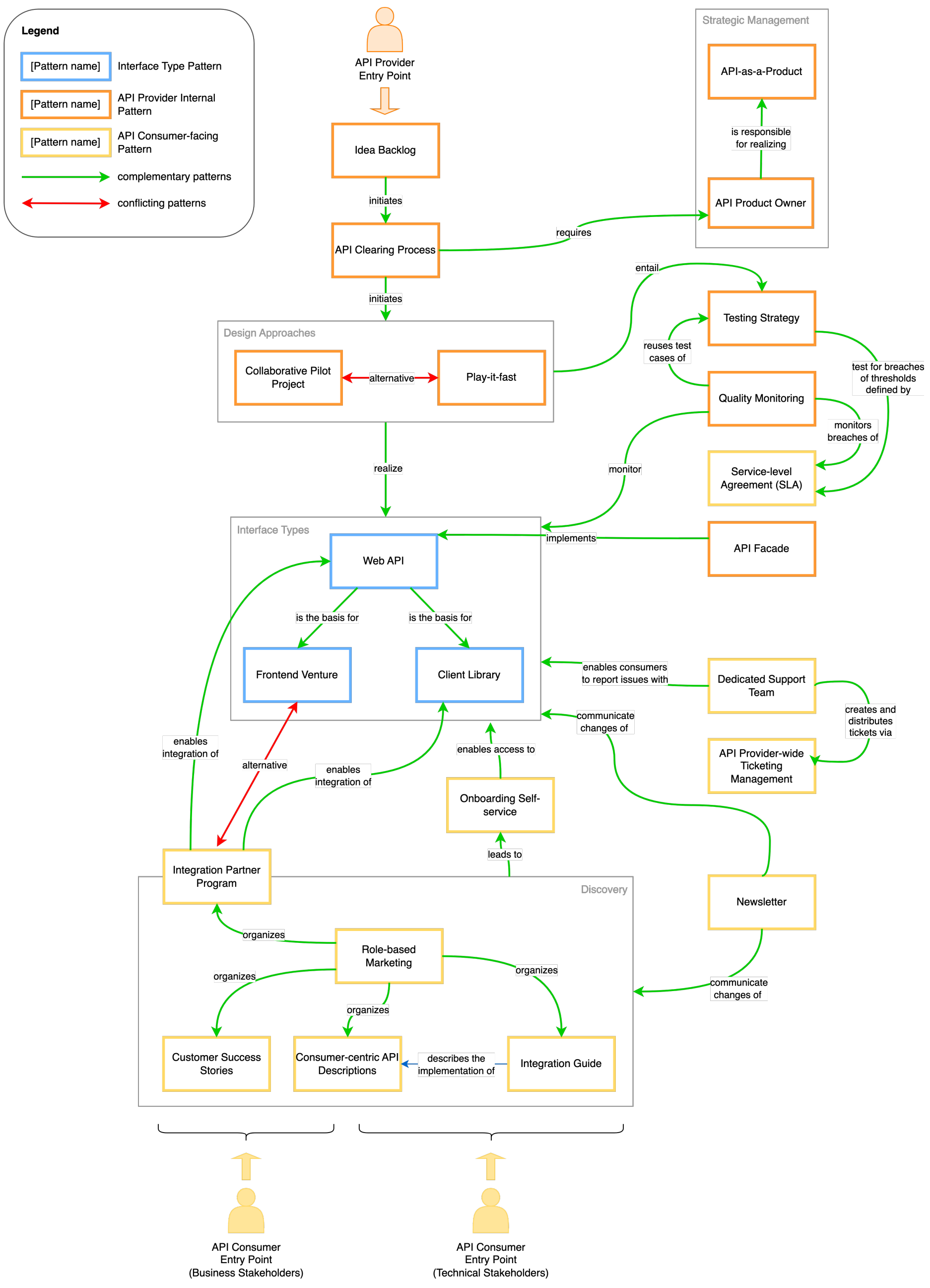

Figure 3.1.: Visualization Web API management pattern relations.

3.3. API Management Pattern Summaries

In the following sections, we present short summaries of each pattern. The overview enables readers to gain a quick understanding of the pattern language and identify patterns relevant to their problem at hand.

3.3.1. Interface Type Patterns Summaries

Table 3.1.: Summaries of the Interface Type Patterns.

Pattern Name	**Pattern Solution Description**
`Web API`	A provider uses a Web Application Programming Interface (Web API) that exposes functionality or data via the public internet, i.e., uses HTTP(S) as communication protocol (Bermbach & Wittern, 2016; De, 2017; Santoro et al., 2019). The Web API decouples the functionalities implementation from the interface that makes it accessible (De, 2017; Medjaoui et al., 2018; Spichale, 2017). Also, the Web API defines the contract for interactions between the backend and the client application (De, 2017; Jacobson et al., 2012).
`Client Library`	A client library wraps a Web API and enables consumers to access it using code in a specific programming language and compliant with a certain framework (De, 2017; C3). Hence, the application consumer does not have to interact with the Web API directly but indirectly through the library functions in the programming language of choice.
`Frontend Venture`	The API provider enables consumers who cannot, for any reason, integrate an API or client library, to still use the functionality or data via a simple frontend, i.e., a website with fields and buttons that trigger API functionality. If enough consumers are interested in the frontend, a product team can take over the development and maintenance of the frontend.

3.3.2. API Provider Internal Patterns Summaries

Table 3.2.: Summaries of the API Provider Internal Patterns.

Pattern Name	Pattern Solution Description
`API-as-a-Product`	The API provider treats APIs like any other consumer-facing (software) product, including technical, business, legal, marketing, and other aspects.
`API Product Owner`	An API product owner is responsible for an API's economic success, designs and evolves the API according to consumers' needs, and represents the API internally.
`Collaborative Pilot Project`	The API provider designs a new API iteratively in close collaboration with one or a limited set of API consumers to increase the likelihood of the API meeting API consumers' needs.
`Play-it-fast Approach`	The API provider designs and publishes an API based on initially provided consumer requirements but without consumer collaboration during API design and implementation (C4) to achieve fast time-to-market.
`Idea Backlog`	An idea backlog is a dynamic list that stores and aggregates consumer wishes for API endpoints derived from consumer support requests, discussions, or surveys.
`Testing Strategy`	A centrally defined testing strategy enforces the testing of new APIs or changes to existing APIs to reduce the likelihood of unexpected behavior of new or changed APIs or backends (C3).
`Data Clearing Process`	A data clearing process ensures that all API endpoints comply with legal and strategic requirements before they are published externally by involving different stakeholders who provide feedback and need to sign off on a new API or the change to an existing API.
`API Facade`	An API facade abstracts the invocation of several backend services into a single API (Gamma et al., 1994). The API facade thereby supports the tailoring of APIs that fit the user stories of the API consumers.
`API Quality Monitoring`	API quality monitoring describes continuously testing an API's non-functional properties to detect anomalies and take countermeasures quickly.

3.3.3. API Consumer-facing Patterns Summaries

Table 3.3.: Summaries of the API Consumer-facing Patterns.

Pattern Name	Pattern Solution Description
`Role-based Marketing`	Role-based marketing denotes the clear separation of marketing material and other consumer-facing resources targeted at different user roles in the developer portal.
`Customer Success Stories`	A customer success story exemplifies an API consumer's successfully finalized use case or product implementation utilizing the provider's APIs (C3) with the aim to demonstrate an API's potential to future consumers.
`Newsletter`	The API provider publishes summaries of changes to existing APIs (De, 2017) and other announcements related to APIs in a newsletter to keep current and potential future API consumers up-to-date.
`Consumer-centric API Description`	The API provider describes the API products functionality from a consumer perspective as use cases or user stories addressing a consumer's business need.
`Integration Guide`	An integration guide documents the implementation of common functionality using step-by-step instructions (Spichale, 2017) to reduce consumers' effort implementing the specific functionality (Medjaoui et al., 2018).
`Onboarding Self-service`	An onboarding self-service automates (parts of) the API onboarding process by allowing API consumers to choose a monetization plan, register a user account, generate authentication credentials, and register a finalized client application without interacting with API provider team members.
`Integration Partner Program`	API providers support API consumers with finding suitable integration partners by creating and maintaining a curated list of potential integration partners that meet specific quality criteria.
`API provider-wide Ticketing Management`	The API provider uses a uniform ticketing system that manages all API-related tickets and is available to all teams involved in API provision. Hence, the ticketing system enables transparency, e.g., on ticket resolution times or recurring issues.
`Dedicated Support Team`	The dedicated support team accepts all API consumers' questions, service requests, and incident reports and immediately answers or resolves low- or medium-complexity tickets. Only high-complexity tickets are forwarded to the respective experts, relieving the API and backend provider teams of a portion of the support activities.
`Service Level Agreement (SLA)`	An SLA is an agreement between two parties that specifies the quality of services, i.e., the APIs' non-functional properties and support service levels, as well as contractual punishments in case of SLA breaches. Hence, an SLA increases API consumers' trust in an API's quality.

CHAPTER 4

Interface Type Patterns

This chapter presents three *Interface Type Patterns*. Interface Type Patterns document the technical components and resources via which the API provider makes data and functionality available to consumers. The API provider should carefully choose between Interface Type Patterns depending on the API provider's strategy and the target consumers' capabilities.

4.1. Web API

Pattern Overview	
Name	`Web API`
Pattern Type	Interface Type Pattern
Summary	A provider uses a Web Application Programming Interface (Web API) that exposes functionality or data via the public internet, i.e., uses HTTP(S) as communication protocol (Bermbach & Wittern, 2016; De, 2017; Santoro et al., 2019). The Web API decouples the functionalities implementation from the interface that makes it accessible (De, 2017; Medjaoui et al., 2018; Spichale, 2017). Also, the Web API defines the contract for interactions between the backend and the client application (De, 2017; Jacobson et al., 2012).

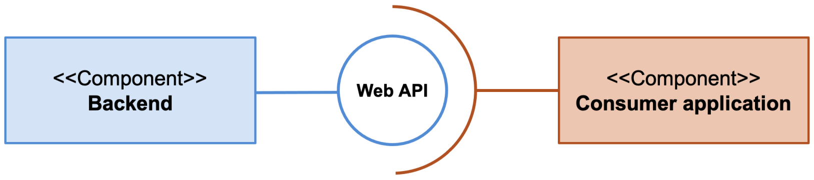

Figure 4.1.: A `Web API` makes functionality and data accessible to API consumers via the HTTP protocol.

Context:

A system provides some valuable assets, i.e., functionality or data. The organization that owns the system wants to make these assets available to other organizations or the public with the goal of enabling platformization (de Reuver et al., 2018; Eaton et al., 2015; Ghazawneh & Henfridsson, 2010, 2013; Karhu et al., 2018), realizing new business models (De, 2017; Medjaoui et al., 2018), enabling efficient partner integration (Jacobson et al., 2012), or achieving compliance (e.g., in banking or automotive) (Bondel et al., 2020).

Concern:

How can an organization (provider) make functionality or data accessible to other organizations (consumers)?

Forces:

- *Stability.* Close coupling between the backend and a client application forces the developers of the client application to change or at least test the integration after each change to the backend (Erl, 2008; Spichale, 2017).

- *Information hiding.* Direct integration between a backend and client application requires that the client application developer knows and understands the backend implementation.
- *Programming language and platform independence.* Direct interaction between a backend and a frontend requires that both software components communicate using the same programming language or platform. If the components do not communicate in the same programming language or use the same platform, the provider or consumer have to implement a wrapper.
- *Security.* Exposing internal assets via the web creates a vulnerability that adversaries could try to exploit (De, 2017).
- *Web API quality.* Badly implemented web API interfaces are difficult to understand and use for API consumers and, thus, lead to increased development costs for API consumers. If API consumers can choose between Web APIs, they may abandon a low-quality Web API (Spichale, 2017).

Solution:

The asset owner implements a Web Application Programming Interface (Web API). Generally, APIs separate the backend implementing a functionality or managing data from the interface that makes these assets accessible to consumers (De, 2017; Medjaoui et al., 2018; Spichale, 2017). Hence, the API is an interface between two software components that define how these components interact, including the data format specification and the protocol used to transport the data (De, 2017; Jacobson et al., 2012). Therefore, an API is often described as a "contract" between two applications (De, 2017; Jacobson et al., 2012). A *Web* API is a specific type of API that exposes its endpoints over the public internet (Santoro et al., 2019), i.e., leverages HTTP as transport protocol for data or as an application protocol that additionally dictates the semantics for API behavior (Daigneau, 2011).

Stakeholders:

The *API provider* has to collaborate with the *backend provider* to make the backend functionality or data accessible via Web API. The *API provider* is responsible for designing, publishing, and maintaining the `Web API`. Furthermore, the *API provider* has to collaborate with the *API consumers* to identify the required level of quality.

Implementation Hints:

Web API types. Web APIs can be categorized into different types depending on the organizational affiliation of the API consumer. First, private APIs describe Web APIs that are used either by internal developers (*internal APIs*) or by a restricted set of partners (*partner APIs*) (De, 2017; Jacobson et al., 2012). In addition, we introduce a third type of private API, *group APIs*, which describe Web APIs that one subsidiary makes accessible to other subsidiaries within a group setting. In comparison, *public APIs* are accessible to every internal and external developer (De, 2017) as long as they adhere to the terms of use (Jacobson et al., 2012) and potentially pay a usage fee.

In this Web API management pattern catalog, we focus exclusively on Web APIs with consumers outside the API provider's organization. Hence, public, partner, and group Web APIs are in scope, while internal APIs are out of scope.

Web API Management. A Web API is a software component that the API provider implements to achieve a business goal (Jacobson et al., 2012; Medjaoui et al., 2018). Hence, the API provider has to manage the Web API throughout its lifecycle (Medjaoui et al., 2018). Such API management comprises, e.g., testing, documenting, publishing, monitoring, securing, and handling changes of the API (Medjaoui et al., 2018) similar to classic software development management.

Architectural styles. API providers can implement Web APIs using different architectural styles. The most prominent architectural style for Web APIs is *REST* since it uses many HTTP built-in functionalities (De, 2017). REST stands for *REpresentational State Transfer* and was published by Fielding (2000).

An alternative to REST are *RPC-style (Remote Procedure Call)* Web APIs (Daigneau, 2011; Maleshkova et al., 2010; Santoro et al., 2019). RPC-style APIs define their own operations, which consumers can invoke using HTTP methods (Maleshkova et al., 2010).

Finally, a relevant architectural style for Web APIs is *SOAP (Simple Object Access Protocol*, often also referred to as *Web Services* (Daigneau, 2011; Santoro et al., 2019; Spichale, 2017). SOAP usually uses HTTP(S) as a communication protocol, but it could also be implemented using another protocol, e.g., SMTP (Spichale, 2017).

Consequences:

Benefits:

- *Stability.* A Web API separates the backend that implements functionality from the Web API interface that makes the functionality accessible to API consumers. Hence, the API provider can make changes to the backend without affecting the client integration as long as the Web API interface model remains the same. As a result, integrations are more stable (Medjaoui et al., 2018; Spichale, 2017).
- *Information hiding.* The separation between the backend and the Web API allows the API provider to hide the details and complexities of the backend implementation from the client developer (Bermbach & Wittern, 2016; Spichale, 2017).
- *Programming language and platform independence.* Web APIs are programming language and platform independent due to their use of the HTTP protocol (Spichale, 2017).
- *Security.* There are many best practices and patterns for ensuring the security of Web APIs (De, 2017; Jacobson et al., 2012; Medjaoui et al., 2018; Spichale, 2017). However, the design and implementation of security measures require effort.

Drawbacks:

- *Stability.* While the API provider can easily change the backend implementation without impacting the API consumer, changes to the Web API itself can lead to the need to adopt client applications (Spichale, 2017). If such changes happen frequently, it can frustrate the consumers (De, 2017). Hence, the API provider has to carefully evaluate, design, implement, and communicate such changes.
- *Web API quality.* API providers have to design, implement, and maintain high-quality Web APIs. This pertains not only to the technical aspects of Web APIs but also to additional resources like documentation and processes like onboarding.

Related Patterns within this Pattern Catalog:

The pattern `Web API` provides the basis for all patterns within this catalog. Therefore, API providers can apply the pattern in conjunction with all other patterns.

Other Related Patterns:

First, Geewax (2021) presents a collection of *API Design Patterns* that document the technical implementation of a `Web API`.

Pautasso et al. (2016) presents patterns describing sequences of interactions between a REST API and a client that realize specific non-functional requirements. An API provider can refer to these patterns when implementing the `Web API` as a REST API.

Similarly, Bellido et al. (2013) documents four types of control-flows for decentralized RESTful service composition with different variations. An API provider can use these patterns when designing a `Web API`.

Daigneau (2011) describes 25 web service design patterns, including web service API styles, client-service interaction styles, request and response management, web service implementation styles, web service infrastructure, and web service evolution patterns. The web service API styles comprise `RPC APIs`, `Message APIs`, and `Resource APIs`. An API provider can use these patterns to implement a `Web API`.

Hohpe and Woolf (2003) presents a pattern language for enterprise integration focusing on asynchronous messaging. Their root pattern `Messaging` is a foundation for `Web API` implementation.

In comparison, Richardson (2019) and Richardson (n.d.) present the pattern `Messaging`, and alternatively, the pattern `Remote Procedure Invocation (RPI)`. While `Messaging` describes asynchronous communication between services, `RPI` captures the concept of a client interacting with a service via a synchronous, remote procedure invocation-based protocol. Hence, both approaches can enable the realization of `Web APIs`.

Similarly, Erl (2008) presents `Service Messaging`. `Service Messaging` captures the approach of exchanging data between applications via *"[...] independent units of communication routed via the underlying infrastructure"* (Erl, 2008, p. 533). Moreover, Erl (2008) emphasizes the need for delivery guarantees, security, state and context management, efficient real-time interactions, and coordination of cross-service transactions to enable SOA based on messaging. Also, several other patterns specialize `Service Messaging` in Erl (2008), e.g., `State Messaging`, `Service Callback`, and `Reliable Messaging`.

Known Uses:

We observed the pattern in 13 cases:

The API initiative of C2 makes simulation and modeling algorithms accessible. The Web APIs follow the REST architectural style.

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops via Web APIs implemented following the REST architectural style.

Cases C4 and C5 represent a software service provider's public and partner API initiatives. End users with a license for the API provider's software product automatically have access to the public API to enable customized system integration. The partner API allows other software providers to create integrations with their software products, thus creating more integrated solutions for end-users. Again, these Web APIs follow the REST architectural style.

In C6, the API provider is a subsidiary within a mobility group that offers API management services. The backend providers and API consumers are both subsidiaries within the same group. The API management service supports API providers with Web APIs' design, implementation, and maintenance.

The organization C7 is an insurance subsidiary that provides insurance services within a group setting. The API management team offers a mix of Web APIs and frontends. If the API provider exposes a Web API or a frontend depends on the nature of the product. The products that need much integration into backend systems, customization, or branding are exposed as Web APIs. Additionally, the API provider provides frontends for products that do not need much integration.

Next, in C9, the organization offers a marketplace for IoT applications. The API provider again offers Web APIs and frontends, however, most consumers prefer frontends. Nevertheless, the API provider also provides Web APIs to enable integration and automation.

The API provider of C10 offers partner Web APIs for easy integration of a financial service into websites and online shops to major clients.

Case C12 captures a financial service provider that provides SaaS software to end-users. The API provider uses Web APIs to enable other software providers to integrate their software with the SaaS system of C12, thus offering an integrated solution to the end users.

In C13, the organization offers a public marketplace for financial services applications. The API marketplace provider offers core platform software, and third-party developers can build additional modules for the platform using Web APIs. Users can buy the core platform software and add pre-integrated modules according to their needs.

C14 captures the Web API of a SaaS software product to which users with a software license automatically have access. Access to the Web API enables the end-users to integrate the software product with their application landscape.

Stripe (C16) provides an API for online payment processing[1]. The Stripe API is a REST API[2]. Stripe also offers several server-side and client-side libraries[3].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[4]. Twilio offers REST APIs[5] and several client libraries for server-side and client-side programming[6].

Cross-case observations:
Since the provision of a Web API is the requirement for inclusion as a case in this Web API management catalog, we listed all cases. There is only one exception, case C11, since the provider decided to exclusively publish an SDK. The included cases vary concerning their scope, size, and monetization.

[1]https://stripe.com/en-de/about (accessed 20.12.2023).
[2]https://stripe.com/docs/api?lang=ruby (accessed 20.12.2023).
[3]https://stripe.com/docs/api?lang=ruby (accessed 20.12.2023).
[4]https://www.twilio.com/ (accessed 20.12.2023).
[5]https://www.twilio.com/docs/usage/api (accessed 20.12.2023).
[6]https://www.twilio.com/docs/libraries (accessed 20.12.2023).

4.2. Client Library

Pattern Overview	
Name	`Client Library`
Alias	Helper Library (Higginbotham, 2016; C16)
Pattern Type	Interface Type Pattern
Summary	A client library wraps a Web API and enables consumers to access it using code in a specific programming language and compliant with a certain framework (De, 2017; C3). Hence, the application consumer does not have to interact with the Web API directly but indirectly through the library functions in the programming language of choice.

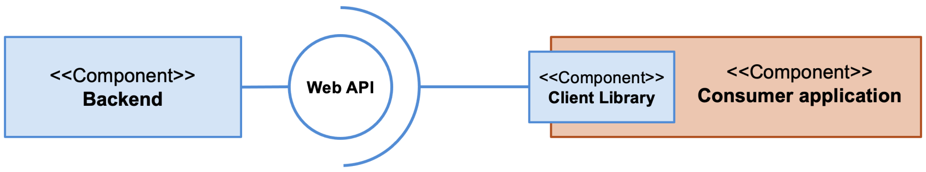

Figure 4.2.: A `Client Library` wraps a Web API so that API consumers can access the API functionality and data using a specific programming language and framework.

Context:

An API provider exposes Web APIs and their documentation. The Web APIs make data and functionality available via the HTTP protocol using different paradigms like REST or data query languages like GraphQL[7]. However, API consumers want to access the data and functionality with their applications written in specific programming languages and potentially using certain frameworks[8] (De, 2017). Thus, each API consumer has to transform the API requests and responses from the HTTP protocol into the respective programming languages or frameworks, resulting in additional effort and longer API adoption times (De, 2017; Spichale, 2017).

[7]https://graphql.org/ (accessed 20.12.2023).

[8]A framework predefines the architecture of an application, including *"[...] the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control."* (Gamma et al., 1994, p. 26). Instead of the application developer implementing the main body of the application and calling libraries or toolkits to reuse existing code, the framework provides the main body and architecture, and the application developer implements the code that the framework calls (Gamma et al., 1994). Thus, a framework *"[...] emphasize[s] design reuse over code reuse"* (Gamma et al., 1994, p. 27).

Concern:

How can API providers enable API consumers to easily integrate an API into applications using different programming languages and frameworks?

Forces:

- *Adoption time.* API consumers want to be able to integrate APIs into their applications quickly. Hence, an API provider should provide resources that accelerate API adoption (De, 2017).
- *Boilerplate code.* API consumers have to write boilerplate code to access Web APIs in specific programming languages.
- *Developer Experience.* A solution should positively affect the developer experience of API consumers.
- *Design and maintenance.* The API provider has limited resources. Thus, the solution should create little additional effort for the API provider organization.
- *API changes.* The API provider changes an API often or less often over time, depending on the provider's chosen change management strategy.
- *Creativity.* A solution should not limit the creativity of API consumers.

Solution:

The API provider offers client libraries as additional resources for the API consumers. Client libraries wrap the Web API and enable API consumers to access the API using code in a specific programming language and compliant with a certain framework (De, 2017; C3). The API provider makes the client libraries available to the API consumer on a developer portal, public repositories, or package managers. In addition to the library code itself, the API provider publishes client library documentation.

Stakeholders:

The *API provider* has to design, publish, and maintain the `Client Library`. However, the API provider has to collaborate with the *API consumers* to ensure that they publish the correct client libraries with the right quality.

Implementation Hints:

Basic idea. The API provider offers a Web API and associated client libraries for specific programming languages and frameworks. A client library is code that transforms HTTP requests into classes or functions and processes HTTP responses in a particular programming language following the language's conventions[9]. Thus, the client library encapsulates the lower-level details of the communication with a Web API. More precisely, client libraries present an object-oriented API to API consumers (Spichale, 2017). API consumers have to download and install the library to use it (Spichale, 2017). Afterward, the API consumer integrates the client library functionality into an application to communicate with the Web API.

Even though out of scope of this pattern, the libraries are sometimes not restricted to transforming the API calls into different programming languages but also offer additional functionality (Gamma et al., 1994) like authentication[10] (C9; C16) or predefined UI components[11] (C16).

Client library types. Client libraries can be either server-side libraries or client-side libraries. Server-side libraries run on the backend server of the API consumers application and support server-side programming languages[12] (C17). Server-side libraries usually cover C#, Java, Node.js, PHP, Python, Ruby, Go, and .NET client libraries[13] (C16; C17).

On the other hand, client-side libraries are processed in a client, e.g., a web browser or a mobile app. Such client-side libraries often cover JavaScript libraries and libraries for mobile devices, e.g., for iOS or Android[14] (C16; C17). Most API providers offer server-side and client-side client libraries[15] (C16; C17).

SDKs. Furthermore, most of the reviewed literature and many API providers use the terms *Software Developer Kit (SDK)*[16] (Spichale, 2017; C16) or *devkit* (De, 2017) in conjunction with client libraries, without clearly defining or delineating the concepts. Generally, the goal of an SDK is to simplify and speed up the integration of a Web API with applications in different programming languages (Higginbotham, 2016; Spichale, 2017). Furthermore, blog posts define different potential components of SDKs, comprising APIs, client libraries, frameworks, documentation, examples scripts for SDK usage, CLI scripts, compilers, and debuggers (Higginbotham, 2016; IBM Cloud Education, 2021; Red Hat, 2020). Hence, we understand an SDK as a set of tools that enable API consumers to use an API with a specific operating system, a specific programming language, or a specific framework. As described in this pattern, a client library consists of code and the associated client library documentation. Therefore, an SDK can comprise a client library.

Documentation. The API provider has to document the client library to enable API consumers to use it, including instructions on how to download and set up the client libraries (De, 2017; Spichale, 2017; C11). Preferably, the documentation includes sample calls that the consumer can use right away (De, 2017; Spichale, 2017). Furthermore, the documentation of

[9]https://cloud.google.com/apis/docs/cloud-client-libraries (accessed 20.12.2023).
[10]https://stripe.com/docs/api?lang=ruby (accessed 20.12.2023).
[11]https://stripe.com/docs/libraries (accessed 20.12.2023).
[12]https://www.twilio.com/docs/libraries (accessed 20.12.2023).
[13]https://www.twilio.com/docs/libraries, https://stripe.com/docs/libraries (accessed 20.12.2023).
[14]https://www.twilio.com/docs/libraries, https://stripe.com/docs/libraries (accessed 20.12.2023).
[15]https://www.twilio.com/docs/libraries, https://stripe.com/docs/libraries (accessed 20.12.2023).
[16]https://www.twilio.com/docs/libraries (accessed 20.12.2023).

a client library should refer to the documentation of the underlying API and the other way around (C3).

We observe that many API providers integrate the documentation of server-side client libraries into the API documentation[17] (C16; C17). The API provider documents the API with use cases according to `Consumer-centric API Description`. The API provider complements each use case step with an HTTP message or curl command code snippet that implements the step. Moreover, the API provider adds code snippets to realize these steps with each of the client libraries. Then, the API consumer can simply choose the programming language of the code snippets[18]. Moreover, the API provider should publish reference documentation following the conventions of the respective programming languages of the client library, e.g., Javadoc[19] for the Java library.

However, in all cases, the documentation of client-side libraries is detached from the Web API documentation and follows documentation conventions of the respective programming languages and frameworks[20] (C16; C17).

Repository. API providers make the client libraries available for download via their developer portal (De, 2017; Spichale, 2017) , public repository platforms, or package managers. For example, many API providers enable API consumers to download client libraries from *GitHub*[21] (C9; C11), *npm*[22] (C16; C17) or PyPi[23] (C17). Also, the API provider can decide to publish client libraries as open-source projects[24] (C11; C17).

Automation. Several tools exist that automatically generate client libraries for different programming languages and frameworks from machine-readable API descriptions (C3; C4; C5). For example, the tools *Swagger Codegen*[25] and *Restlet Studio*[26] can generate client libraries based on OpenAPI specifications[27] (De, 2017). Similarly, *Postman*[28] enables a user to generate code snippets for 29 programming language and framework combinations[29] from postman collections[30]. The tools *APIMatic.io*[31] and *REST United*[32] can generate SDKs using RAML[33] descriptions (De, 2017). Also, tools exists for generating SDKs from Blueprint[34] (De, 2017).

[17]https://www.twilio.com/docs/libraries, https://stripe.com/docs/libraries (accessed 20.12.2023).
[18]https://www.twilio.com/docs/usage/api (accessed 20.12.2023).
[19]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html (accessed 20.12.2023).
[20]https://stripe.com/docs/libraries (accessed 20.12.2023).
[21]https://github.com/ (accessed 20.12.2023).
[22]https://www.npmjs.com/ (accessed 20.12.2023).
[23]https://pypi.org/project/pip/ (accessed 20.12.2023).
[24]https://www.twilio.com/docs/libraries/go (accessed 20.12.2023)
[25]https://swagger.io/tools/swagger-codegen/ (accessed 20.12.2023).
[26]https://restlet.talend.com/ (accessed 20.12.2023).
[27]https://swagger.io/specification/ (accessed 20.12.2023).
[28]https://www.postman.com/ (accessed 20.12.2023).
[29]https://learning.postman.com/docs/sending-requests/generate-code-snippets/ (accessed 20.12.2023).
[30]https://www.postman.com/collection/ (accessed 20.12.2023).
[31]https://www.apimatic.io/ (accessed 20.12.2023).
[32]https://restunited.com/ (accessed 20.12.2023).
[33]https://raml.org/ (accessed 20.12.2023).
[34]https://apiblueprint.org/ (accessed 20.12.2023).

The API provider can generate client libraries, potentially enhance them, and offer them to API consumers. Alternatively, the API provider can make structured API descriptions accessible to API consumers, thus enabling the API consumers to auto-generate the client libraries themselves (C3; C4; C5). However, manually designed client libraries allow API providers to integrate domain knowledge into the libraries and prevent biases imposed by automation tools, e.g., biases regarding preferred API styles (Medjaoui et al., 2018).

Know your audience. The creation and maintenance of client libraries create additional effort for the API provider. Therefore, the API provider should only provide client libraries for which a demand exists. Hence, the API provider needs to know the API consumers and what programming languages and frameworks they prefer (De, 2017; Spichale, 2017; C4; C5).

Change. As soon as the API implements a breaking change, the API provider has to create new versions of all client libraries wrapping an API (Medjaoui et al., 2018; C4; C5; C16). For every change to the client library, the API provider has to update the related documentation and inform users. In addition, the API provider has to deprecate the old client library version. Thus, the API provider should define a change management strategy, e.g., only offering the latest version of a library, thus forcing the API consumers to adapt their application with each breaking update. Alternatively, the API provider can maintain several versions of the same library (C9; C17).

Client libraries only. In most cases, the API provider offers direct access to an API and, in addition, client libraries. However, we also observed one case where the API provider gives access to an API exclusively via a client library (C11).

Consequences:

Benefits:

- *Adoption time.* The provision of client libraries relieves consumers from writing wrappers. Thus, libraries allow API consumers to integrate Web APIs faster and thus reduce adoption times (De, 2017; Spichale, 2017). Thus, API consumers with specific programming language expertise do not need to understand or learn how to access the raw Web API or transform the API requests and responses (C2).
- *Boilerplate code.* Client libraries reduce the need for API consumers to write boilerplate code to access Web APIs using specific programming languages[35].
- *Developer Experience.* Since the consumers do not need to spend time implementing boilerplate code to access the API[36], a client library can positively affect the developer experience.

Drawbacks:

[35]https://cloud.google.com/apis/docs/cloud-client-libraries (accessed 20.12.2023).
[36]https://cloud.google.com/apis/docs/cloud-client-libraries (accessed 20.12.2023).

- *Design and maintenance.* The creation and maintenance of client libraries lead to additional effort for the API provider (Spichale, 2017; C4; C5). The API provider has to create the API manually, or automatically with a tool and potentially enhance it. Moreover, the API provider must create documentation and link or integrate it with the Web API documentation (De, 2017; Spichale, 2017). In addition, the API provider has to manage changes to the client library and the respective documentation (Medjaoui et al., 2018). Finally, the API provider should manage a client library similar to a product, which includes marketing the library and providing support to API consumers (De, 2017; C4; C5). Due to the high effort connected to the provision of client libraries, few internal APIs offer client libraries (Spichale, 2017).
- *API changes.* Changes to an API can lead to a ripple effect that requires adjusting client libraries. Thus, especially in settings where APIs change frequently, it can be effortful to update client libraries constantly. Moreover, in addition to the client libraries themselves, the API provider has to adapt the client libraries' documentation in case of changes to the libraries (Medjaoui et al., 2018). Thus, overall, the cost of changing client libraries and other supporting assets can account for a significant amount of an API product's overall change cost (Medjaoui et al., 2018). However, the API provider can take measures to minimize the effort necessary to adapt client libraries in case of changes to the API, e.g., using tools and automation (Medjaoui et al., 2018). A prerequisite for automation is that the API provider uses machine-readable API specifications (Medjaoui et al., 2018). However, Medjaoui et al. (2018) also points out that even though tools for automated code generation support the API provider in improving the developer experience, they also introduce biases, e.g., with regards to API styles. Moreover, the API provider should ensure the tooling does not create a lock-in effect (Medjaoui et al., 2018).
- *Creativity.* The client libraries encapsulate the web API, making it easier to use the API in a programming language but also imposing a certain structure. Thus, client libraries can hinder the creativity and freedom of API consumers (Dal Bianco et al., 2014). However, if the API provider additionally makes the API itself accessible, API consumers can nevertheless use all the APIs functionality.

Related Patterns within this Pattern Catalog:

An API provider can publish a `Client Library` in conjunction with a `Web API` to allow API consumers to easily integrate an API into applications using different programming languages and frameworks. In one case (C11), the API provider even decides to offer only the `Client Library` as an interface for API consumers and not make the `Web API` accessible.

Also, the API provider can realize a `Client Library` through a `Collaborative Pilot Project` or a `Play-it-fast Approach`. Furthermore, `API Quality Monitoring` can monitor the breach of non-functional properties of a `Client Library`. The `Dedicated Support Team` enables consumers to report issues related to `Client Library` and a `Newsletter` can communicate changes of a `Client Library`. Finally, an `Onboarding Self-service` enables access to a `Client Library`.

Other Related Patterns:

A `Client Library` is a specific implementation of an `Adapter` as presented by Gamma et al. (1994). The `Adapter` pattern captures the general concept of converting an existing interface into another (domain-specific) interface to meet clients' expectations.

Known Uses:

We observed the pattern in five cases:

The API initiative of C2 makes simulation and modeling algorithms accessible. The API provider additionally offers a client library to relieve the API consumers from making REST HTTP calls.

In C9, the organization offers a public marketplace for IoT applications. The organization provides a core platform software, and third-party developers can build additional modules for the platform using APIs. API consumers can interact with the core platform software using RESTful APIs. In addition, the API provider offers client libraries that allow API consumers to interact with the API in different programming languages. These libraries often also offer additional functionality, e.g., easy authentication. All libraries are available for download on the API provider's developer portal, and some libraries are also available on a public repository. For some libraries, different versions exist, and the API consumers are advised to use or upgrade to the latest versions. Furthermore, the API provider refers to open source libraries created by the developer community.

The API provider of C11 offers a financial service for API consumers to integrate into their systems. The API itself is not exposed, but the API provider publishes an SDK in a very popular programming language. The API provider documents the SDK, including sample calls and release notes.

Stripe (C16) provides an API for online payment processing[37]. The Stripe API is a REST API[38] but Stripe offers several server-side libraries for programming languages, including Ruby, Python, and Java. The documentation of these client libraries is integrated with the documentation of the Stripe REST API [39]. Additional client libraries for JavaScript frameworks include a wrapper for the API and predefined user interface (UI) components[40]. Also, Stripe publishes client libraries for mobile platforms, including iOS and Android[41]. These JavaScript and mobile client libraries follow documentation conventions of the respective programming languages and frameworks[42]. Most of the client libraries are open source and available on

[37]https://stripe.com/en-de/about (accessed 20.12.2023).
[38]https://stripe.com/docs/api?lang=ruby (accessed 20.12.2023).
[39]https://stripe.com/docs/invoicing/connect (accessed 20.12.2023).
[40]https://stripe.com/docs/libraries (accessed 20.12.2023).
[41]https://stripe.com/docs/libraries (accessed 20.12.2023).
[42]https://stripe.com/docs/libraries (accessed 20.12.2023).

GitHub[43] or npm[44]. Finally, Stripe also refers to open source libraries created by the Stripe community[45].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[46]. Twilio offers REST APIs[47] and several client libraries for the server-side programming languages C#, Java, Node.js, PHP, Python, Ruby, and Go[48]. The API consumer can download the client libraries using different package managers, e.g., Maven to download the Java helper library[49] or PyPi to download the Python library[50]. Some of the client libraries are open source, e.g., the Go library[51]. Twilio integrates the documentation of these client libraries in the REST API documentation[52]. Moreover, Twilio offers different versions of client libraries, but new features and bug fixes are only added to the latest version of each library[53]. Also, Twilio offers client-side libraries, including iOS, Android, and JavaScript SDKs[54], and refers to open-source libraries developed by external developers[55]. Finally, Twilio publishes machine-readable OpenAPI specifications for the APIs, enabling API consumers to use tooling that automatically generates client libraries in many more programming languages[56].

Cross-case observations:

Four out of the five API initiatives are very similar. These cases describe public initiatives in production with high numbers of consumers. Furthermore, the providers use APIs as the main or only distribution channel for a software product. In addition, these initiatives have predefined plans or pay-per-call rates to monetize access to their APIs. Therefore, it makes sense that the providers invest much effort into making the APIs accessible to a broad audience.

Case C2 is an outlier. The partner initiative is in a pilot stage, has few consumers, and does currently not monetize the use of the API. However, we assume that the provider wants to attract first users and thus provides simple libraries.

[43]https://github.com/stripe (accessed 20.12.2023).
[44]https://www.npmjs.com/package/stripe (accessed 20.12.2023).
[45]https://stripe.com/docs/libraries (accessed 20.12.2023).
[46]https://www.twilio.com/ (accessed 20.12.2023).
[47]https://www.twilio.com/docs/usage/api (accessed 20.12.2023).
[48]https://www.twilio.com/docs/libraries (accessed 20.12.2023).
[49]https://mvnrepository.com/artifact/com.twilio.sdk/twilio (accessed 20.12.2023).
[50]https://pypi.org/project/twilio/ (accessed 20.12.2023).
[51]https://www.twilio.com/docs/libraries/go (accessed 20.12.2023).
[52]https://www.twilio.com/docs/usage/api (accessed 20.12.2023).
[53]https://www.twilio.com/docs/libraries/java/usage-guide-8 (accessed 20.12.2023).
[54]https://www.twilio.com/docs/libraries (accessed 20.12.2023).
[55]https://www.twilio.com/docs/libraries/community-supported-libraries (accessed 20.12.2023).
[56]https://www.twilio.com/docs/libraries (accessed 20.12.2023).

4.3. Frontend Venture

A previous version of this pattern has been published in Bondel et al. (2022). In this pattern catalog, we evolved the pattern.

Pattern Overview	
Name	`Frontend venture`
Pattern Type	Interface Type Pattern
Summary	The API provider enables consumers who cannot, for any reason, integrate an API or client library, to still use the functionality or data via a simple frontend, i.e., a website with fields and buttons that trigger API functionality. If enough consumers are interested in the frontend, a product team can take over the development and maintenance of the frontend.

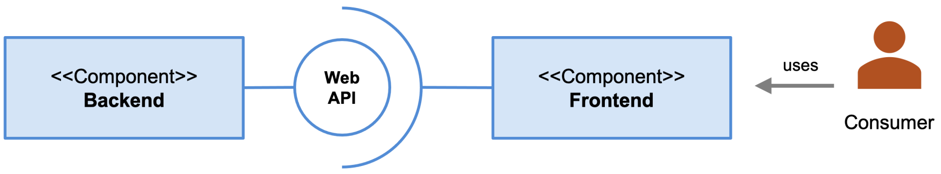

Figure 4.3.: A `Frontend Venture` makes data and functionality available to consumers lacking the capabilities to use a `Web API` or `Client Library`.

Context:

Integrating an API into an existing IT landscape creates effort for the API consumers. However, some API consumers lack technical capabilities or the budget for API integrations and can thus not realize beneficial use cases. Such consumers are often municipalities or small, non-digital businesses (C3).

Concern:

How can an API provider enable API consumers that cannot, for any possible reason, integrate an API to use the API's functionality or data nonetheless?

Forces:

- *Consumer capabilities.* Some potential consumers, especially municipalities or small, non-digital organizations, are not able to use or integrate APIs since they have no or small IT departments that are already working at capacity. For example, in one observed case, the API consumers are small, non-digital organizations that usually only have a website based on a content management system maintained by freelancers (C3).

Furthermore, these potential consumers often have no budget to hire external IT service providers to execute the integration project (C3).

- *Profits.* API consumers are willing to pay for solutions that meet their needs.
- *Public perception.* The API provider can make data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest. Such initiatives are viewed positively by the public (C3).
- *Legal obligation.* An API provider can be legally obliged to make specific data available to certain consumer groups through APIs, even if some consumers cannot use the API (C3). Such legal obligations exist, for example, in the banking or automotive industry.
- *Effort.* API providers often work at capacity and consumers use a software solution only if the API provider can provide it with a certain level of quality.
- *Reusability.* Solutions always need to balance the specific needs of the first API consumer or end user with the reusability of the solution for other consumers.
- *Flexibility.* An API and an interface provide different flexibility with regards to integration (C9), automatizing (C9), customization (C7), and branding (C7).

Solution:

The API provider identifies and evaluates a use case with an API consumer or a consumer group and implements a frontend, i.e., a website with fields and buttons that trigger API functionality. As soon as the frontend reaches a certain level of maturity, the API provider markets it to other potential consumers. If enough consumers are interested, a product team takes over the development and maintenance of the product. Thus, implementing a frontend can be a venture opportunity.

Stakeholders:

The *API provider* has to design and implement the frontend. During the design and implementation, the API provider has to collaborate with the *API consumers* to ensure that the frontend meets the consumers' needs concerning functional and non-functional requirements. Furthermore, the API provider should aim to hand over the responsibility for the frontend to a dedicated *product team* after its publication.

Implementation Hints:

Basic approach. A frontend implements a specific use case for a consumer or consumer group. Therefore, the first step is to analyze the need of the future consumer or consumer group. Once the use case is defined, the API provider assesses the benefits and drawbacks. The benefits can include additional direct profits from billing the consumers, the chance of further profits from reselling the frontend to other external consumers, internal reuse of the frontend, or positive marketing impact (C3). The API provider has to weigh these advantages against the additional effort required to develop and maintain the frontend.

Furthermore, as part of the initial analysis, the API provider should check if similar frontends exist within the organization. If an internal team has already built a similar frontend, the API provider can reuse (parts of) it (C3). Based on this benefit-cost evaluation, the API provider decides if or how to implement the frontend.

Assuming the API provider decides to move forward with the frontend implementation, in the next step, the API provider designs and implements the frontend. The API provider can choose to employ either a `Collaborative Pilot Project` or a `Play-it-fast Approach` to design and implement the frontend. An alternative approach would be to implement a simple first version of the frontend in the course of a `hackathon`[57].

After the frontend reaches a certain level of maturity, the API provider can present it to other interested parties. If enough consumers are interested in using the frontend, the API provider can hand it over to a dedicated product team to evolve and maintain it as a product following the `API-as-a-Product` pattern (C3). Thus, designing and implementing a frontend is a venture opportunity for the providing organization. The product team then acts as an IT provider to the consumer while the API platform provides the underlying API services (C3).

Types of frontends. A frontend realizes a specific use case and typically concentrates on a subset of the APIs functionality. In general, the goal is to provide a user interface using state-of-the-art design elements that consumers without technical capabilities can easily and intuitively use. The most basic type of a frontend is a user interface that simply makes the as-is API functionality usable for non-technical consumers. For example, the API provider can implement a website that allows users to upload data, set transformation parameters and response filtering options, and subsequently download a file containing the APIs response (C3; C7). A more advanced frontend can also implement some logic that augments the response data with additional data or visualizations. As an example, a frontend can show a map and locate certain events on it (C3).

Consequences:

Benefits:

- *Consumer capabilities.* The frontend allows API providers to make API data and functionality available to organizations with insufficient IT capabilities or budgets. Still, other interested API consumers with enough IT capabilities can consume the API to create custom integrations or proprietary frontends.
- *Profits.* The API provider can monetize the implemented frontend, and it can thus become a source of profit. If the API provider does not provide the frontend, a third party could skim these profits.

[57] A hackathon is an event with a pre-defined timeframe during which small development teams compete to implement the best solution to a pre-defined problem.

- *Public perception.* Making data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest results in positive publicity for the API provider (C3).
- *Legal obligation.* In case of a legal obligation, the API provider has to implement the API anyway. Therefore, the API provider might view the legal obligation as an opportunity to create new business relationships or profits through the frontend (C3).

Drawbacks:

- *Effort.* The design, implementation, and especially the maintenance of the frontend create additional effort for the API provider. If the frontend is of insufficient quality, the API consumers will not use it, and it would thus be a loss of investment (C3). The API provider has to ensure that enough resources are available to realize a frontend venture.
- *Reusability.* The API provider has to put effort into balancing the needs of different frontend consumers (C3). If the API provider tailors the frontend too much to the need of one API consumer, other API consumers will not use the frontend. However, if the frontend is too generic, it might be of less value for all consumers.
- *Flexibility.* A frontend limits the consumers flexibility compared to an API with regards to integration (C9), automation (C9), configuration (C7), and branding (C7). Thus, it is important that the API provider also keeps providing the API.

Related Patterns within this Pattern Catalog:

An API provider can design and publish a `Frontend Venture` in addition to a `Web API` if, for any possible reason, the consumer cannot integrate a `Web API` (C3). However, the provider should also publish the `Web API` to preserve the flexibility of consumers that want to integrate it directly (C7; C9).

Also, the API provider can realize a `Frontend Venture` through a `Collaborative Pilot Project` or a `Play-it-fast Approach`. Furthermore, `API Quality Monitoring` can monitor the breach of non-functional properties of a `Frontend Venture`. The `Dedicated Support Team` enables consumers to report issues related to `Frontend Venture` and a `Newsletter` can communicate changes of a `Frontend Venture`. Finally, an `Onboarding Self-service` enables access to a `Frontend Venture`.

Finally, the pattern `Frontend Venture` is suitable if the consumer has neither the technical nor the financial capabilities to integrate a `Web API`. However, suppose the consumer lacks only technical capabilities but has a sufficient budget for API integration. In that case, the API provider can alternatively use a `Integration Partner Program` to introduce integration partners to the consumer.

Known Uses:

We observed the pattern in three cases:

The API portal provider of an automotive organization (C3) wanted to provide data to two municipalities via APIs. However, the municipalities did not have the capabilities to integrate the APIs and requested a user interface that a non-technical stakeholder can use. The API provider implemented such an interface during a pilot project since the organization expected positive marketing effects. After the pilot phase, the API provider team handed the prototype over to a product team that now maintains the frontend. As a result, the API provider reports that only 50% of consumers directly access the API behind the frontend, while 50% use the frontend.

The organization C7 is an insurance subsidiary that provides insurance services within a group setting. The API management team offers a mix of APIs and frontends. If the API provider exposes an API directly or uses a frontend depends on the nature of the product. The products that need much integration into backend systems, customization, or branding are exposed as APIs. Additionally, the API provider provides frontends for products that do not need much integration. Those are primarily products offering only data access or simple functionality.

Finally, in C9, the organization offers a marketplace for IoT applications. The API provider again offers APIs and frontends, however, frontends are dominant. According to the interview partner, most consumers prefer frontends. Nevertheless, the API provider also provides APIs to enable integration and automation.

Cross-case observations:

All the cases are in early phases (pilot phase or early production phase). This makes sense, since it can be beneficial for API initiatives in early stages to implement frontends to attract first consumers. However, it is also essential to maintain the Web APIs since Web APIs provide more flexibility regarding backend integration or frontend customization to organizations with more IT capabilities or budget.

Also, not surprisingly, the consumers are rather small business or municipalities.

CHAPTER 5

API Provider Internal Patterns

This chapter presents nine *API Provider Internal Patterns*. API Provider Internal Patterns require the API provider to collaborate mainly with stakeholders internal to the API provider organization.

5.1. API-as-a-Product

Pattern Overview	
Name	`API-as-a-Product (AaaP)` (Medjaoui et al., 2018)
Alias	Tailoring APIs into Services (C5)
Pattern Type	API Provider Internal Pattern
Summary	The API provider treats APIs like any other consumer-facing (software) product, including technical, business, legal, marketing, and other aspects.

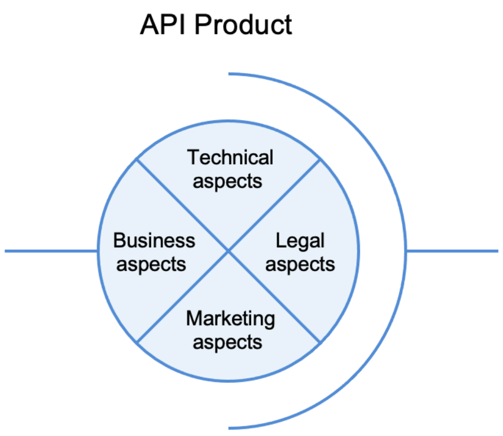

Figure 5.1.: When applying the pattern `API-as-a-Product`, an API provider has to take technical, business, legal, and marketing aspects into consideration for each Web API.

Context:

An API provider wants to launch an API initiative by exposing existing APIs or creating new APIs for external API consumers. Successful API initiatives have to solve the specific business needs of API consumers. However, the API provider does not know how to design APIs that meet such consumers' business needs.

Concern:

How can an API provider design APIs that meet the business needs of potential API consumers?

Forces:

- *Business needs.* API consumers use APIs that address their business needs (ITIL 4, 2019). Also, consumer needs change over time. If consumers do not use an API, its design and development results in a waste of resources (ITIL 4, 2019).
- *Product management.* The identification and definition of new processes, methods, and techniques for managing APIs create effort for API providers. A solution should enable API providers to employ proven product management methods.
- *Perception.* An API is a consumer-facing piece of software that influences the perception of the API provider's organization and the relationship with customers.
- *Quality.* API consumers use APIs only if they meet their quality expectations. Furthermore, APIs with low quality can result in high maintenance and evolution costs for the API provider (ITIL 4, 2019).
- *Developer experience.* The developer experience plays an important role in API consumers' choice of an API. The developer experience describes the experience of an API consumer interacting with an API to achieve a certain outcome (ITIL 4, 2019). Thus, a solution should take into account the functional and non-functional characteristics of an API and additional aspects that influence the API consumers' ease of interaction with the API, e.g., access, support, documentation.
- *Unanticipated innovation.* A goal of APIs is to enable unanticipated innovation (Jacobson et al., 2012). Hence, in addition to defined use cases, an API should foster unforeseen innovation through its consumers.
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.

Solution:

An API product is a bundle of endpoints that the API provider manages like any other (software) product, including business, financial, legal, and marketing aspects (Medjaoui et al., 2018). The API provider presents a Web API that solves a business need and defines a monetization scheme to create an API product. Furthermore, the provider can differentiate API products into tiers based on non-functional characteristics, e.g., the number of calls, support service levels, or analytics services.

Stakeholders:

The *API provider* has to collaborate with the *upper management* to identify relevant consumer groups. Moreover, the API provider should collaborate with *sales & marketing*, *finance & controlling*, and *legal* to align the marketing, business model, and legal considerations for an API product. Also, the API provider has to communicate with the *backend providers* to adapt or create and bundle endpoints into API products. Finally, the API provider should closely collaborate with *API consumers* to identify their needs and design suitable API products.

Implementation Hints:

API Product characteristics. An API product denotes an API or a set of APIs that the API provider treats like any other (software) product (Medjaoui et al., 2018). An API product has the following characteristics:

- *Business need.* An API product addresses an external consumer's business need (C6) and thus creates value. Instead of simply making an existing functionality available to external API consumers, the API provider identifies use cases or user stories that the API provider can realize with an API (C3; C6). For example, instead of providing an API that allows to "send emails," an API provider can address the user story "notify customers that a package has been delivered" (C6).
- *Strategic goal:* An API product has a strategic goal derived from the organization's overall business strategy (Jacobson et al., 2012; Medjaoui et al., 2018). Strategic goals are, for example, the improvement of customer reach or the acceleration of time-to-market (De, 2017; Medjaoui et al., 2018). Furthermore, the API provider monitors KPIs to analyze if the API products reach their strategic goals (De, 2017; Jacobson et al., 2012).
- *Endpoint bundles.* Instead of publishing single endpoints, the API provider combines API endpoints into a bundle. An API endpoint combines related endpoints or endpoints that together enable a more complex functionality (De, 2017; C3; C10; C13).
- *Business models and monetization.* An API product has a business model (C6). A business model balances the expected benefits of publishing an API product with the investment made to design and manage it (De, 2017). The business model can rely on direct or indirect monetization, depending on the strategic goal of the API product (De, 2017). In the case of direct monetization[1], the API provider can choose between flat-rate subscription, usage-based pricing, and auction-style allocation approaches (Stocker et al., 2018). In many cases, API providers also use a freemium model that allows new API consumers to use the API for free up to a specific rate limit or point in time (Stocker et al., 2018).
- *Portfolio management.* The API provider considers dependencies of API products on the overall product landscape of the organization (ITIL 4, 2019).
- *Lifecycle.* API products have a lifecycle. The API provider actively manages the API products along the lifecycle (C6) from inception and design through maintenance to retirement (ITIL 4, 2019; Medjaoui et al., 2018).

[1]A more detailed overview of possible monetization schemes can, for example, be found here: https://medium.com/@ama.thanu/what-are-the-different-api-business-models-9709ae45f416 (accessed 20.12.2023).

- *Product Team.* Each product has a product team (C13), including an `API Product Owner` (De, 2017).
- *Mindset.* An API provider organization fosters an internal API product mindset, i.e., the understanding of APIs as strategic products that must be managed accordingly (C6; C9). Furthermore, the API provider acknowledges the ongoing relationship between the API provider and the consumers (Medjaoui et al., 2018).
- *Product marketing.* An API provider employs product marketing approaches to gain API consumers' interest. Such marketing approaches comprise campaigns and advertising at events and organizations (C3).Also, the API provider should target business and technical API consumers with targeted marketing measures, e.g., by applying `Role-based Marketing` (C3).
- *Support.* The API provider organization provides support for API consumers in the same way the organization supports consumers of other (software) products (C14). Either the product team provides consumer support, or the API provider uses the pattern `Dedicated Support Team`, transferring some support tasks from the product team to a dedicated support team.
- *Security concept.* The API provider has a security concept in place for API products just like for other (software) products the organization offers (C14).
- *Discoverability.* The API provider aims to ensure that consumers can discover and understand the value of API products (C3; C13). In some cases, the API providers use `Consumer-centric API Description` to support discoverability.
- *Product tiers.* An API provider can offer different product tiers (De, 2017). Each product tier makes the same functionality available but under different conditions (De, 2017). Such different conditions can comprise different rate limits[2] (De, 2017),pricing (C11), terms and conditions (C14), SLAs (Stocker et al., 2018), rights regarding allowed operations (read-only vs. read-write) (De, 2017), or support services. Different product tiers address the needs of different types of API consumers, e.g., private or business users (C3). In many cases, the API consumer can choose product tiers on the API developer portal in a self-service fashion (De, 2017).
- *Change.* API products are part of an open, distributed system and require change over time (Lübke et al., 2019). The API provider actively manages change and acts on consumers' feedback (Medjaoui et al., 2018).
- *Product families.* In cases in which APIs are not stand-alone products but part of a landscape, they can be viewed as part of a *"product family"* (Medjaoui et al., 2018). APIs that are part of a product family should implement similar usage patterns since it allows

[2]A rate limit defines how many calls an API consumer can make to an API within a specific time interval depending on the API product (tier) (De, 2017; Jacobson et al., 2012).

developers working with several API products to benefit from similarities (Medjaoui et al., 2018).

However, in practice, we observed that not all API products completely fulfill these characteristics. For example, the interview partners mentioned several times that the API product mindset still needs improvement. Similarly, in the case base, we rarely observed the definition of a strategic goal for each API and the monitoring of goal achievement.

API product design. An API product must address a consumer group's business need to succeed (ITIL 4, 2019). Thus, before creating an API product, the API provider has to identify a target consumer group (Jacobson et al., 2012; Medjaoui et al., 2018), which is potentially derived from the organization's overall strategy (Medjaoui et al., 2018). For example, the API provider could use an end-user or developer segmentation analysis[3] in combination with force-ranking[4] priority segments to identify relevant target groups (Jacobson et al., 2012).

Given the target consumer group, the API provider has to analyze the consumers' business needs and design products that address these business needs (Medjaoui et al., 2018). Potential approaches for identifying consumer needs and useful API products are the Value Proposition Canvas[5] (Osterwalder et al., 2014) or Design Thinking[6] (ITIL 4, 2019; Medjaoui et al., 2018). Additionally, the concept of *"Jobs To Be Done"*[7] (JTBD) can help API providers to design consumer-centric API products (Medjaoui et al., 2018).

In the case base, we also observed "Domain Story Telling"[8] and "Event Storming"[9] as approaches to identify functionality with value for API consumers (C6).

Finally, since designing products that meet the consumers' and providers' goals is challenging, the provider should employ iterative and incremental approaches to developing such

[3]Market segmentation describes the process of categorizing consumers within a market into groups according to characteristics, e.g., behavioral or geographic characteristics. Market segmentation enables targeted marketing, and advertising (Tynan & Drayton, 1987).

[4]Force-ranking describes the ranking of segments against each other to identify the most important segments.

[5]The value proposition canvas is a tool to systematically define a value proposition, i.e., the benefits a solution creates for a customer or customer group. The value proposition canvas relates customer jobs, pains, and potential gains to the provider's solution. It forces the provider to clarify how the solution relieves pains and creates gains for the consumer. Thus, it tries to ensure that a provided solution meets a need of the target consumer (Osterwalder et al., 2014).

[6]Design thinking is a method or mindset that aims to create innovative solutions for complex problems using diverging thinking and an iterative and end user-oriented approach (Uebernickel & Brenner, 2016).

[7]"Jobs-to-be-done" (JTBD) describe what a customer tries to achieve in a particular circumstance. Thus, JTBD describe the causal driver for a buying decision instead of focusing on correlations between a customer's characteristics and a purchase (Christensen et al., 2016).

[8]The goal of domain storytelling is to transfer domain knowledge from domain experts to the team responsible for designing and implementing business software, e.g., developers, testers, product owners, product managers, and business analysts (Hofer & Schwentner, 2021). Domain storytelling is a workshop format during which a domain expert tells a story, i.e., a real example, from their domain. In parallel, the workshop moderator visualizes the story as a diagram. All participants observe how the story and visualization unfold, thus preventing misunderstandings and detecting issues.

[9]Event Storming describes a workshop format in which participants collectively describes changes to the state of a domain, i.e., domain events, to gain a better understanding of the domain and identify or solve related problems (Brandolini, 2013). The approach results in models that support domain-driven design.

products (ITIL 4, 2019).

However, we mainly observed less structured approaches to API product design. In many cases, the API providers simply design API products based on consumer feedback or requests. For example, the pattern `Idea Backlog` allows API consumers to communicate ideas for new APIs or changes to existing APIs to the API provider (C13). A commonality between most cases is that the API providers implement APIs only if at least one external consumer specifically requests them (C11; C13). Also, close collaboration with API consumers and feedback cycles should be part of identifying functionality (C6).

Moreover, API providers can hire design agencies to support user research and user experience design (C13).

After identifying an API product, the API provider implements it. The API provider can apply incremental and iterative methods to the API development process (ITIL 4, 2019). Moreover, the API provider can use a `Collaborative Pilot Project` to ensure that the product meets the consumers' requirements (C11; C13). In addition to the technical implementation of the API product itself, the API provider has to consider the needs of the target consumer concerning the user experience, i.e., the ease of use and other non-functional requirements (ITIL 4, 2019).

Consequences:

Benefits:

- *Business needs.* The design and development of APIs using approaches like the Value Proposition Canvas (Osterwalder et al., 2014), Design Thinking (Medjaoui et al., 2018; Uebernickel & Brenner, 2016), JTBD (Medjaoui et al., 2018), Domain Story Telling (Hofer and Schwentner, 2021; C6), and Event Storming (Brandolini, 2013; C6) increases the likelihood of an API meeting consumer needs. Furthermore, the iterative and incremental evolution of APIs (ITIL 4, 2019) enables the API provider to adopt the API product to changing consumer needs. Moreover, the API provider combining several endpoints into an API product relieves API consumers from identifying and subscribing to separate endpoints that address a business need together. In summary, offering an API product that addresses business needs makes it easier for the API consumer to discover, commit to, and integrate the APIs.

- *Product management.* Bundling APIs into products enables the API provider to apply proven (software) product design, development, and marketing techniques (C3).

- *Perception.* Treating APIs as products ensures that API providers are aware of the influence of APIs on how consumers perceive the organization.

- *Quality.* Treating APIs as products instead of mere technical interfaces leads to increased prioritization of API quality (C9).

- *Developer experience.* API products aim not only to meet the consumers' needs with regards to functional and non-functional requirements but also regarding business and other aspects, e.g., support services. Moreover, the design of API as products considers the overall developer experience for API consumers and aims to achieve a positive experience.

Drawbacks:

- Business needs. The API provider should be aware that the success of an API strongly depends on the attractiveness of the assets that the APIs make accessible. Thus, creating an API product does not lead to the success of an API initiative if the assets are not attractive for API consumers in the first place (Jacobson et al., 2012).
- Unanticipated innovation. The design of API products that address specific use cases of certain consumer groups might limit the potential for unanticipated innovation, especially if the API provider combines APIs into bundles and makes them accessible only as part of such bundles.
- Effort. The API provider has to identify, design, and maintain API products that meet the consumers' needs. Due to the continued collaboration with API consumers and the involvement of other departments, API product management can lead to much effort for the API provider.

Related Patterns within this Pattern Catalog:

The pattern `API-as-a-Product` documents an overarching concept of API management that guides the design, implementation, evolution, and maintenance of an API. A provider who chooses to apply the pattern `API-as-a-Product` should also appoint an `API Product Owner` responsible for the APIs realization (De, 2017; Medjaoui et al., 2018; C6; C11; C14).

Other Related Patterns:

In Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018), the authors describe the pattern `Pricing Plan`. Pricing allows the API provider to charge API consumers for using an API product. A pricing plan can follow a subscription-based, usage-based, market-based allocation, or freemium approach.

Similarly, Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018) present the pattern `Rate Limit`. A rate limit enforces a limit for API calls according to a specific API product tier.

Hence, the patterns `Pricing Plan` and `Rate Limit` are more specialized patterns that can support the realization of certain aspects of the pattern `API-as-a-Product`.

Next, Medjaoui et al. (2018) presents the concept "API-as-a-product," which means *"We treat our APIs just like any other product we offer"* (Medjaoui et al., 2018, p. 39) including *"[...] proper design thinking, prototyping, customer research, and testing, as well as long-term monitoring and maintenance."* (Medjaoui et al., 2018, p. 39). In addition, Medjaoui et al. (2018) emphasizes the management of API products along their lifecycle and introduce ten pillars representing the API product management work domains. The pillars are Strategy, Design, Documentation, Development, Testing, Deployment, Security, Monitoring, Discovery, and Change Management.

Finally, ITIL 4 (2019) defines products as *"A configuration of an organization's resources designed to offer value for a consumer"* (ITIL 4, 2019, p. 12). These resources comprise people, information, technology, value streams and processes, and partners and suppliers. An organization designs products to meet the needs of one or a set of specific consumer groups. Moreover, consumers usually only see the components of a product necessary to meet these needs.

Services build on products and are *"A means of enabling value co-creation by facilitating outcomes that customers want to achieve, without the customer having to manage specific costs and risks"* (ITIL 4, 2019, p. 12). A *service offering* is a description of one or more services (ITIL 4, 2019). Service offerings can include goods transferred to the consumer, access to resources under the provider's control, and agreed-on service actions, e.g., support services. An organization can make the same product available as different service offerings with varying conditions to several consumer groups. For example, a provider can make a software service available as a limited free version and a full paid version.

Known Uses:

We observed the pattern in seven cases.

The API portal provider of an automotive organization (C3) makes vehicle data accessible to businesses and private consumers. The organization bundles API endpoints into products and provides descriptions for business and technical stakeholders. Moreover, the API provider describes the products using API consumer use cases. Finally, the organization uses marketing campaigns to market the API products.

In C6, the API gateway provider is a subsidiary within a mobility group that offers API management services. These services comprise the management of a central API gateway and the consultation of subsidiaries during API development. In some cases, the API gateway provider even implements the APIs for these other subsidiaries. The API gateway provider does not speak about *"API products"* but instead uses the term *"digital services"* to talk about the same concept. A digital service combines a technical interface with a use case, a business model, and lifecycle management. The API gateway provider collaborates with the other subsidiaries to identify relevant use cases that create consumer value. The gateway provider uses Domain Storytelling and Event Storming workshops to identify these use cases.

The API provider of C11 offers financial services for API consumers to integrate into their systems. The organization invests much effort into an API product's initial design to reduce changes after the publication to a minimum.

In C13, the organization offers a marketplace for insurance services. The API marketplace provider allows third-party API providers to publish their API products on the marketplace. In addition, the API provider organization offers its own API products on the marketplace.

These API products are developed in close collaboration with future API consumers. Also, each API product documentation describes the product's business value.

C14 is a financial services provider that offers software operated in the cloud to end users. APIs enable users to integrate the software into their system landscapes. Additionally, third-party software provider organizations can create integrations with their software products, thus offering an already integrated solution to the end-users. The API provider treats the API like any other software product. Each product has its own support and security concept. Moreover, a central governance unit governs all API products.

Stripe (C16) provides an API suite for online payment processing and related services[10]. Stripe lists several API products, e.g., the product "Online payments"[11]. The online payments product comprises several use cases, for example, "Accept online payments" or "Create a subscription"[12].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[13]. Twilio groups products into categories, e.g., the category "Channel APIs"[14]. Within the channel API category, Twilio groups APIs into products according to communication channels, e.g., into the group "Programmable Messaging"[15]. Furthermore, Twilio defines fees for the usage of the different APIs[16].

Cross-case observations:

We observed API providers using API products in the context of public or group API initiatives. Most cases reflect more mature API initiatives with several APIs in production and several existing consumers. Added to this, most organizations are software or IT service providers. Thus they have experience with designing and managing intangible products.

[10]https://stripe.com/en-de/about (accessed 20.12.2023).
[11]https://stripe.com/docs/products (accessed 20.12.2023).
[12]https://stripe.com/docs/payments (accessed 20.12.2023).
[13]https://www.twilio.com/ (accessed 20.12.2023).
[14]https://www.twilio.com/products (accessed 20.12.2023).
[15]https://www.twilio.com/products (accessed 20.12.2023).
[16]For example https://www.twilio.com/sms/pricing/de, https://www.twilio.com/whatsapp/pricing/de, and https://www.twilio.com/conversations/pricing (accessed 20.12.2023).

5.2. API Product Owner

Pattern Overview	
Name	`API Product Owner`
Alias	Product Manager (Jacobson et al., 2012), API Product Manager (Medjaoui et al., 2018)
Pattern Type	API Provider Internal Pattern
Summary	An API product owner is responsible for an API's economic success, designs and evolves the API according to consumers' needs, and represents the API internally.

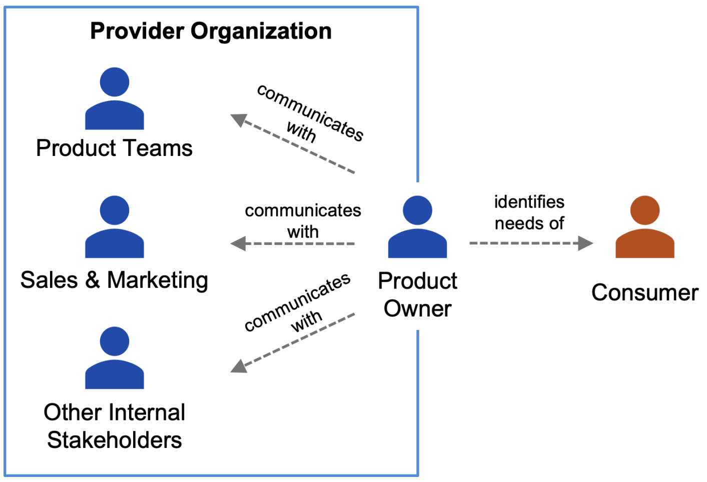

Figure 5.2.: An `API Product Owner` connects knowledge about the API provider organization's internal processes with knowledge of API consumers' needs.

Context:

An API provider wants to expose a new or existing API externally. The API provider treats the API as an `API-as-a-Product` that needs to meet the API consumers' needs to be successful. Thus, managing APIs requires knowledge of external API consumers' needs and API provider organization internal capabilities and processes to design and implement a successful API.

Concern:

How can an organization realize API management that focuses on API consumers' needs?

Forces:

- *Required knowledge.* A solution needs to bring together internal knowledge of an organization's capabilities and processes with the understanding of external API consumers' needs to design feasible and demanded APIs.
- *Introduction effort.* The API provider should be able to quickly and easily introduce the solution with as little change to the organizational structure as possible.

Solution:

An organization can realize consumer needs-driven API management by introducing an API product owner (API PO) role for each API. An API product owner is responsible for investigating consumers' needs and ensuring that new APIs or changes to existing APIs meet these needs, thus enabling economic success. Furthermore, the product owner represents and drives the design and implementation of APIs internally.

Stakeholders:

The API product owner as part of the *API provider* team has to collaborate with all stakeholders interested in the API. This includes the collaboration with the *Backend providers* to identify and access existing functionality or initiate the implementation of new functionality. Furthermore, the API product owner has to collaborate with the *API consumers* to collect feedback and keep track of their needs.

Implementation Hints:

Role and responsibilities. An organization appoints an API product owner for each API or API product (C5; C6; C12; C14). The API product owner belongs to the API provider organization and thus knows internal capabilities and processes (C5; C6; C12; C14). At the same time, the API product owner is responsible for the economic success of the API throughout its lifecycle (Medjaoui et al., 2018) by packaging these APIs into a product (De, 2017; Medjaoui et al., 2018; C14). Thus, an API product owner needs to combine the understanding of the organizational context and the API product domain (Medjaoui et al., 2018).

Tasks. The API product owner can have several responsibilities and tasks. The following list of responsibilities is non-exhaustive:

- *Understand consumer needs.* An API product owner has to understand and collect the consumers' needs (De, 2017; C12). Hence, an API product owner regularly talks to existing and potential future API consumers to identify their pains, needs, and wishes (C5). The person taking over the API product owner role should be able to understand, simplify, and prioritize consumer requirements (Jacobson et al., 2012), and speak the API consumers' language (C14). Finally, based on the knowledge from discussions with API consumers, the API product owner creates use cases that address existing and potential API consumers' pains, needs, and wishes (C5).

- *Decision-making.* An API product owner is responsible for making decisions on implementing new APIs (C5) and evolving existing APIs (C5; C11).
- *Business goal definition.* A task of the API product owner is to define business goals, and KPIs for an API (Medjaoui et al., 2018).
- *Business case calculation and funding.* Additionally, an API product owner is responsible for analyzing the expected profitability of a new or changed use case (C6). To do so, the API product owner calculates business cases and defines monetization schemes (De, 2017). If a use case is profitable, the API product owner initiates all necessary internal processes to request funding for new APIs or changes to existing APIs (C3; C6).
- *Translate business requirements.* The API product manager translates business requirements into technical requirements (De, 2017; Medjaoui et al., 2018).
- *Developer experience.* The API product owner accounts for the API consumers' developer experience (Medjaoui et al., 2018), including the definition of design rules and processes for app approval (De, 2017).
- *Consumer relation management and communication.* The API provider maintains relations with the API consumers (C5; C12; C14). Additionally, the API product owner communicates new APIs or changes to existing APIs to consumers (C12).
- *Internal representation and collaboration.* The API product owner represents the API or API product within the API provider organization. For example, the API product owner initiates necessary review processes, e.g., by a control board (C14), and represents the API throughout these processes (C3; C9; C12). Also, the API product owner collaborates with internal stakeholders like sales and marketing and coordinates work with the backend teams (De, 2017).
- *Product roadmap.* An API product owner designs a strategy and realization plan or roadmap that drives API changes (Jacobson et al., 2012; Medjaoui et al., 2018). These realization plans or roadmaps must fit the organization's goals (Medjaoui et al., 2018).
- *Monitoring of competition.* The API product owner keeps an eye on competing APIs and defines measures to keep the own API competitive (C11).
- *Central point of contact.* The API product owner is the central point of contact for all concerns related to the API (Medjaoui et al., 2018).

The API product owner shares many of these tasks and responsibilities with other roles. For example, an architect can support the decision on API designs (C5) or a portfolio manager can help calculate business cases (C6).

Consequences:

Benefits:

- *Required knowledge.* On the one hand, the `API Product Owner` belongs to the API provider's organization and thus knows the API provider's internal capabilities and processes. On the other hand, one of the API product owner's main tasks is communicating with API consumers and identifying their needs. Hence, an API product owner combines internal and external knowledge to design feasible and demanded APIs.
- *Introduction effort.* The API product owner role is a concept used in agile development. Introducing an API product owner role in an organization that already applies agile development practices should be easy since the necessary structures and mindset are already in place.

Drawbacks:

- *Introduction effort.* The concept of an API product owner is part of agile development practices. Thus, it can be effortful for organizations using non-agile development practices to introduce an API product owner role.

Related Patterns within this Pattern Catalog:

The `API Product Owner` is the role responsible for *productizing* an API (De, 2017; Medjaoui et al., 2018). Hence, the introduction of an `API Product Owner` supports the pattern `API-as-a-Product` (C6; C11; C14).

In addition, the provider can make it a requirement to appoint an `API Product Owner` as part of the `API Clearing Process` process.

Known Uses:

We observed the pattern in four cases:

The partner API initiative of the software and IT service provider in C5 allows users of their software product to integrate said software with the consumers' system landscape easily. An API product owner and an integration architect are responsible for the API's evolution. The API product owner talks to consumers and collects feedback to identify new functionality and use cases.

Case C6 describes an API initiative within a group setting. The API provider organization is a subsidiary that manages the API platform and supports other subsidiaries with API design, provision, and consumption. As soon as a subsidiary proposes a new API, the API provider's portfolio management team calculates the API's profitability. Suppose the API provider organization expects an API drawing on a subsidiaries' backends to return profits in the future. In that case, the API provider funds the API's design, implementation, and maintenance and assigns an API product owner. However, if the API provider does not expect the API to return profits but the backend provider wants to implement the API anyways, the API provider organization requires the subsidiary to fund the API and appoint an API product owner from the subsidiary organization. Nevertheless, the central API provider supports the API product owner with the API design and implementation. Either way, each API has to have an API product owner.

The API provider of C11 offers a financial service for API consumers to integrate into their systems. The organization assigns a product team to each API. Each API product team has an API product owner who identifies and decides on implementing API changes. In addition, API product owners have to monitor competing APIs and ensure the competitiveness of the organization's APIs.

C12 is a financial services provider that provides software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering an integrated solution to the end users. A dedicated product team, including an API product owner, manages the API by applying agile development practices. The API product owner's tasks include, for example, the collection of feedback from consumers and communicating planned changes to consumers.

C14 is a financial service that offers software operated in the cloud to end users. APIs enable other software providers to integrate their software with the system of C14, thus offering an integrated solution to the end users. Again, dedicated teams are responsible for the evolution of the APIs, each including an API product owner and using agile practices. The API product owners must understand their consumers, know their pains, and implement solutions to these pains. API product owners must constantly adapt their understanding of the pains and evolve the APIs to address them. Furthermore, an API management program can identify the need for new APIs. However, the API management program can only initiate a project once an API product owner is assigned.

Cross-case observations:

The cases represent public, partner, and group API initiatives. Further, three API initiatives are in production, and only one is in the pilot stage. Also, all but one initiative uses developer portals, and the remaining one is a backend API. However, the API initiatives differ regarding monetization, including contractual agreements, free use as part of a product license, completely free use, and pay-per-call. Finally, the size of the API initiatives varies from small (<20) to moderate (<100) to a large (<10,000) numbers of API consumers. Hence, API providers can apply the pattern to API initiatives with different characteristics.

However, all the organizations have in common that they already use agile development practices. This makes sense since the role of a product owner is a concept introduced by agile development approaches.

5.3. Collaborative Pilot Project

Pattern Overview	
Name	`Collaborative Pilot Project`
Pattern Type	API Provider Internal Pattern
Summary	The API provider designs a new API iteratively in close collaboration with one or a limited set of API consumers to increase the likelihood of the API meeting API consumers' needs.

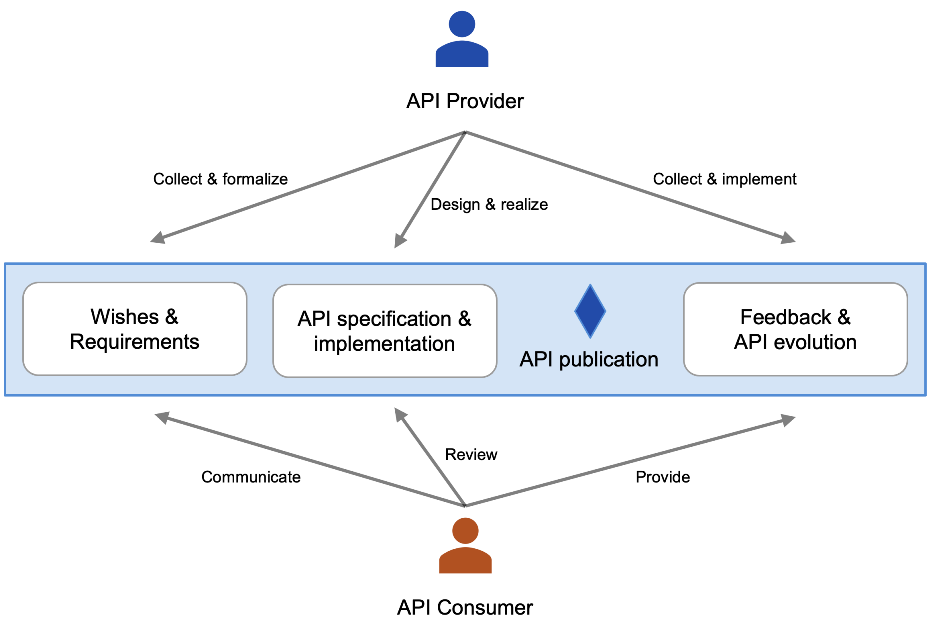

Figure 5.3.: The pattern `Collaborative Pilot Project` involves one or a selected number of API consumers in all steps of the API design. This includes the API consumer reviewing and providing feedback on the API's specification and the prototypical implementation before publication to all consumers.

Context:

An API provider wants to create a new API. For an API to succeed, the APIs must meet the API consumers' needs. Especially for complex use cases, it can be challenging to understand the API consumers' needs correctly (C2).

Concern:

How can the API provider ensure that a new API meets the consumer's needs?

Forces:

- *Consumer demand.* The API provider needs to ensure that APIs meet the consumers' needs so that consumers adopt the APIs. The creation of an API that does not meet consumer needs leads to a loss of investment (C3).
- *Time-to-market.* In specific markets, i.e., high-speed markets, API providers must publish APIs as fast as possible to gain or maintain a competitive advantage (C3). Moreover, even outside of high-speed markets, API consumers that rely on the API to run their business can be negatively affected by long waiting times for new APIs (Medjaoui et al., 2018).
- *Monetization.* The API provider might want to create a new API only if its future monetization is secured. The API provider can ensure future API monetization by negotiating contracts with future API consumers before the API implementation.
- *API stability.* API consumers rely on the stability of the APIs that they integrate. Frequent changes, especially breaking changes, creates effort for API consumers (C2) and can lead to them abandoning an API (Medjaoui et al., 2018).
- *Generic API design.* In public API and marketplace settings, the API provider wants to design APIs to meet the needs of several API consumers.

Solution:

The API provider designs the API iteratively in close collaboration with one or a limited amount of API consumers in a pilot project. The partnering API consumer communicates needs and requirements and provides feedback to the API provider before and during the API implementation project (C3). Hence, the API provider can easily realize changes to the API design before its publication, i.e., before the API is accessible to and integrated by consumers (Medjaoui et al., 2018).

Stakeholders:

The *API provider* has to collaborate with the partnering *API consumer(s)* to discuss and test API implementations during the pilot phase. Usually, these collaborations are based on a contractual agreement. Thus, the provider has to involve the *legal* team. Also, the API provider has to collaborate with the *backend provider(s)* who provide the functionality accessible via the API.

Implementation Hints:

Partner identification. A collaborative pilot project starts with identifying a use case and a consumer who wants to partner. The API provider can identify use cases and partnering consumers during informal direct discussion or via account management or other consumer-facing teams. Alternatively, the API provider can use a more structured process, e.g., the use of an `Idea Backlog`. However, it is also possible that an API consumer actively approaches an API provider and proposes a pilot project. In all cases, the API provider has to evaluate the business case before agreeing to the partnership (C3).

For public APIs, the API provider can also approach top developers with respected positions in specific developer communities. These developers' involvement in the design of new APIs enables improvements of the API and free marketing through evangelization (Jacobson et al., 2012).

Collaboration mode agreement. The partners must negotiate the legal basis for the project. Also, the pilot project partners have to formalize the collaboration and agree upon general conditions, including the mode and intervals of collaboration. These topics can be discussed during a `Pilot Workshop`, which launches the pilot project.

Collaboration modes. The API provider can collect feedback using lab-based usability tests, focus groups, surveys, and interviews (Medjaoui et al., 2018).

Generally, the API consumer can provide feedback at some or all of the following stages of the collaborative pilot project. First, the API provider and consumer can create, discuss, and agree upon API specifications before the API implementation (C3). For example, mocks form a basis to discuss and review APIs (Spichale, 2017). Also, API consumers can review and test the API implementation at specific points during the API implementation, e.g., after each iteration in an agile context (C2; C3). Finally, the consumer can perform the final acceptance test before an API is released into production and potentially made accessible to other external consumers (C2). Furthermore, the API provider can also request feedback from the API consumer on additional artifacts, like the developer portal or API documentation (C2).

Consequences:

Benefits:

- *Consumer demand.* Collecting consumer feedback before an API's publication reduces the need to make assumptions about the API design (Medjaoui et al., 2018) by creating a better understanding of API consumers and their needs. Thus, the chances of API adoption increase. Also, close collaboration creates API consumer buy-in.
- *Monetization.* An API consumer is more likely to sign a contract to use and pay for an API before its implementation if they are closely involved during the APIs design and implementation phases. Hence, a `Collaborative Pilot Project` increases the likelihood of an API consumer agreeing to an upfront contractual monetization agreement.
- *API stability.* It is harder to change APIs after their publication when API consumers have already built integrations (Medjaoui et al., 2018). Iterative improvement of an API based on consumer feedback before publication should reduce the need to make changes after

publication. A stable API with few changes affecting consumer integrations prevents a negative impact on API consumer satisfaction.

Drawbacks:

- *Consumer demand.* API providers sometimes have to implement APIs for legal reasons. In such settings, the consumers' demand for such APIs is a secondary concern for the API provider (C3).
- *Time-to-market.* The API provider depends on the partner providing feedback while testing a new API design. API consumers can have other priorities in their daily business and may not provide feedback to the API provider in a timely manner. Thus, close collaboration with an API consumer can lead to waiting times for the API provider team (C4). Such waiting times disrupt the development processes of the API provider, and postpone planned API go-live dates (C4).
- *Generic API design.* Close collaboration with just one API consumer during the design of an API can lead to a specialized API. Potentially, only one or a small group of API consumers can use an overfitted API (C4). A specialized API is suitable if the API provider aims for a point-to-point integration but not if the API provider wants to provide a public API or a platform in a developer ecosystem that many API consumers can integrate (C12). Finally, close collaboration with third parties comes with the risk of scope creeping (Kendrick, 2015).

Related Patterns within the Pattern Catalog:

A `Collaborative Pilot Project` is an approach to realize a new `Web API`, `Frontend Venture`, or `Client Library` that passed the `API Clearing Process`. The provider can apply a `Testing Strategy` for testing as part of the `Collaborative Pilot Project`.

A `Collaborative Pilot Project` presents an alternative approach to designing and implementing a new API compared to the `Play-it-fast Approach`. While `Collaborative Pilot Project` increases the likelihood of a new API meeting consumer needs due to close and ongoing collaboration, it also increases time-to-market. In comparison, a `Play-it-fast Approach` enables fast time-to-market for new APIs but also increases the risk of not meeting API consumer needs.

Known Uses:

We observed the pattern in three cases:

In case C2, the API provider offers simulation and modeling algorithms for the analysis of energy data. The API provider initiates a new project only if a concrete demand exists, i.e., if an API consumer agrees to partner with them during a pilot project. The API provider gives the collaborating partner access to the developer portal and asks them for feedback. Based on the feedback, the API provider evolves the API endpoint.

The API portal provider of an automotive organization (C3) shares anonymized road condition data with public authorities. The API provider partners with API consumers in pilot projects to better understand the market demand for a new API. Different stakeholders can trigger new projects, but primarily external organizations with prior relations to the provider organization contact the API provider team with new use cases. Furthermore, the API provider aims to design a first prototype of the API endpoint to show to the consumer as fast as possible. Based on the prototype, which can be nothing more than a handwritten specification in the first iteration, the API provider discusses the API endpoint design with the consumer. However, the API provider admits that this kind of close collaboration is not always possible for each API endpoint. Instead, in some cases, the API provider also applies the `Play-it-fast Approach`, i.e., develops and releases an endpoint based on only one prior discussion with a potential API consumer to realize a faster time-to-market.

Case C12 captures a financial services provider that provides SaaS software to end-users. APIs enable other software providers to integrate their software with the SaaS system of C12, thus offering an integrated solution to the end users. The API provider closely collaborates with consumers when creating new APIs for the software product since API consumers often have very concrete expectations regarding the API. However, this approach leads to the creation of several APIs for the same software product, i.e., one API for each consumer.

Cross-case observations:

These cases represent API initiatives still in a pilot phase or in production. The API initiatives are partner and public API initiatives but have a relatively small number of API consumers. This makes sense since collaborative pilot projects usually focus on the specific requirements of single API consumers.

Further, the API providers apply `Collaborative Pilot Project` for API initiatives with other business organizations as well as with government institutions.

5.4. Play-it-fast Approach

Pattern Overview	
Name	`Play-it-fast Approach`
Pattern Type	API Provider Internal Pattern
Summary	The API provider designs and publishes an API based on initially provided consumer requirements but without consumer collaboration during API design and implementation (C4) to achieve fast time-to-market. to enable fast time-to-market.

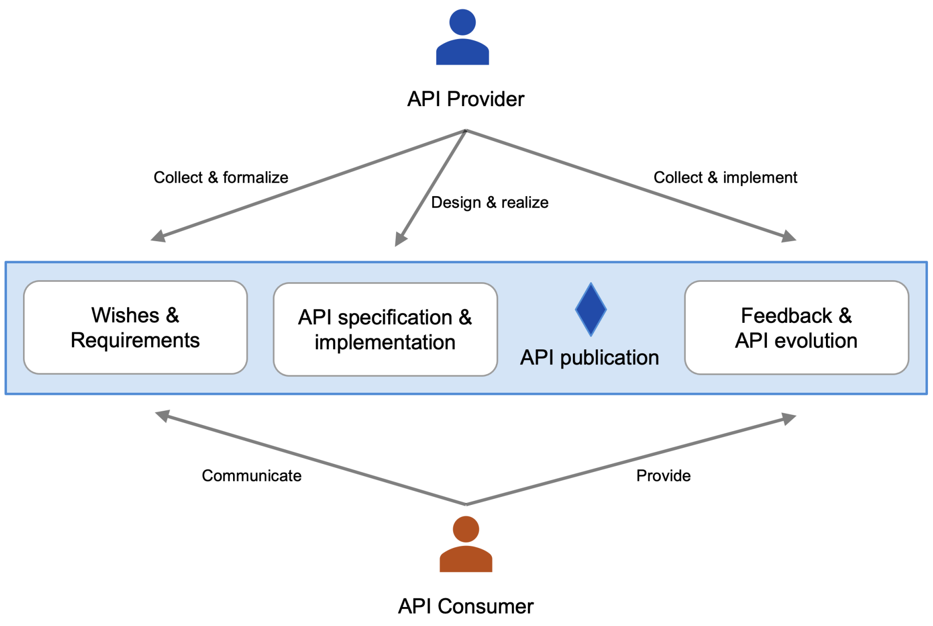

Figure 5.4.: If an API provider applies the `Play-it-fast Approach` pattern to design a new API, API consumers are not involved in the API specification and implementation.

Context:

An API provider wants to publish a new API as fast as possible since a fast time-to-market is of the essence to ensure a competitive advantage (C3). Such a fast time-to-market is relevant, for example, in high-speed markets. Alternatively, even in environments that are not high-speed markets, long waiting times for new APIs can negatively affect API consumers that rely on the API to run their business (Medjaoui et al., 2018). At the same time, the API must meet API consumers' needs.

Concern:

How can an API provider publish new APIs in a fast manner?

Forces:

- *Time-to-market.* In specific markets, i.e., high-speed markets, API providers must publish APIs as fast as possible to gain or maintain a competitive advantage (C3). Moreover, even outside of high-speed markets, API consumers that rely on the API to run their business can be negatively affected by long waiting times for new APIs (Medjaoui et al., 2018).
- *Consumer demand.* The API provider needs to ensure that APIs meet the consumers' needs so that consumers adopt the APIs. Creating an API that does not meet consumer needs leads to a loss of investment (C3).
- *Generic API design.* In public API and marketplace settings, the API provider wants to design APIs to meet the needs of several API consumers. A generic API design allows different consumers to use an API (C12).
- *Monetization.* The API provider might want to create a new API only if its future monetization is secured. The API provider can ensure future API monetization by negotiating contracts with future API consumers before the API implementation.
- *API stability.* API consumers rely on the stability of the APIs that they integrate. Frequent changes, especially breaking changes, creates effort for API consumers (C2) and can lead to them abandoning an API (Medjaoui et al., 2018).

Solution:

The API provider collects the API consumers' requirements for a new API but designs and implements it in isolation (C4). Afterward, the API provider publishes the finalized API to all API consumers, i.e., makes the API accessible and allows consumers to integrate it (Medjaoui et al., 2018). API consumers can now provide feedback on the published API (C2). Based on consumer feedback, the API provider iteratively improves the published API (C2; C4; C12).

Variants:

The API provider can also collect the API consumers' requirements for a new API, design and implement it, and publish the API in a sandbox to which only a limited number of selected consumers have access (C4). This approach is also called a *beta release* (De, 2017). In such settings, the API consumers with access to the sandbox provide feedback to the API provider before the API is published to all API consumers. However, the API provider has to give the API consumers with access to the beta version enough time to test it and provide feedback (De, 2017). Hence, the time until the API provider can publish the API to all consumers, and therefore time-to-market, increases. Nevertheless, the number of required changes to the API after its publication likely decreases.

Stakeholders:

The *API provider* has to initially collect the requirements of the *API consumer(s)* but does not engage with them during the API design and implementation. However, the provider has to collaborate with the *backend provider(s)* to design and adapt the APIs.

Implementation Hints:

Aggregate wishes. The API provider should collect the wishes of API consumers to identify the need for a new API (C4). The pattern `Idea Backlog` enables the collection and aggregation of consumer wishes.

API and documentation. The API provider should not only iteratively improve the API design based on consumer feedback but also the API documentation (C2).

Communication channels. The API provider must provide suitable communication channels that allow API consumers to provide feedback easily after publishing a new API (C2; C12). Furthermore, the API provider needs to ensure that API consumers are aware of these communication channels.

Feedback filter. The API provider has to review received feedback and distinguish between useful and useless remarks (C12).

Consequences:

Benefits:

- *Time-to-market.* The API provider decouples the implementation of an API from the API consumers' feedback allowing for a fast time-to-market (C4; C12). In comparison, if the API provider chooses an approach that requires regular feedback from an API consumer, e.g., by applying the pattern `Collaborative Pilot Project`, the API provider might has to wait for the API consumer's feedback for a long time (C4; C12).
- *Generic API design.* The API provider does not closely collaborate with an API consumer, reducing the risk of overfitting a new API to the needs of just one or a small group of API consumers (C4). Hence, the approach is especially suitable for public and marketplace APIs.

Drawbacks:

- *Consumer demand.* A solution following the `Play-it-fast Approach` pattern collects the API consumers' requirements and wishes at the beginning of the implementation project and involves the API consumer only after the API's initial publication. Hence, the API provider has to make a lot of assumptions on API design (Medjaoui et al., 2018). As a result, there is a risk of implementing an API that does not meet the API consumers' needs, leading to a loss of investment.

- *Monetization.* It is unlikely that an API consumer will sign a contract to use and pay for an API before its implementation without close collaboration during the APIs design and implementation phases. Hence, the API provider has to carry the initial project costs and the risk of API consumers not using the API after publication.
- *API stability.* It is more difficult to change an API after its publication to the API consumers (Medjaoui et al., 2018). Publishing an API without collecting consumer feedback can result in the need to make more changes after publication. Frequent changes in published APIs will likely negatively affect API consumers' satisfaction with the API provider. In some cases, frequent changes lead to API consumers abandoning an API (Medjaoui et al., 2018).

Related Patterns withing this Pattern Catalog:

A `Play-it-fast Approach` is an approach to realize a new `Web API`, `Frontend Venture`, or `Client Library` that passed the `API Clearing Process`.

The provider can apply a `Testing Strategy` for testing during the `Play-it-fast Approach`.

In addition, it is an alternative approach to designing and implementing new APIs compared to `Collaborative Pilot Project`. While the application of `Play-it-fast Approach` enables fast time-to-market for new APIs, it increases the risk of not meeting API consumer needs and the need to make changes to the API after its publication. In comparison, `Collaborative Pilot Project` increases the likelihood that the API design meets API consumer needs but increases the time until a provider can publish an API since the provider has to wait for the consumer to review intermediate API specifications and designs.

Known Uses:

We observed the pattern in three cases.

In the case of C2, the API provider offers simulation and modeling algorithms to analyze energy data. The API provider implements and publishes new APIs based on consumer feedback but without the consumer's involvement during the API implementation. API consumers can access the published APIs via `Onboarding Self-service` and provide feedback to the API provider. Since the API has a limited number of users and the API provider knows all API consumers, the API consumers provide their feedback via Microsoft Teams or Email. The feedback triggers further API changes and improvements to API documentation.

Case C4 represents a software service provider's public API initiative. End users with a license for the API provider's software product automatically have access to the public API to enable customized system integration. The API provider reported that they initially tried involving specific API consumers in the design of APIs according to the pattern

`Collaborative Pilot Project`. However, the approach disrupted the running sprints since the API provider had to wait for feedback from API consumers. In addition, the API provider wants to provide generic APIs that large groups of API consumers can use, but the close collaboration with one API consumer bears the risk of designing APIs specific to specific cases of the partnering API consumer. Hence, the API provider decided to use the `Play-it-fast Approach` pattern. First, the API provider identifies generic use cases that meet the needs of many consumers via the `Idea Backlog`. Then, the API provider designs and implements a new API or change to an existing API and releases it in a sandbox environment to which API consumers have access. The API consumers provide feedback to the API provider as soon as the API or API changes are released. The API provider reports that in the past, they only had to make small changes to APIs after the initial release.

Finally, case C12 captures a financial services provider that provides software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering an integrated solution to the end users. The API provider decides on a project-by-project base if they apply the `Play-it-fast Approach` or the `Collaborative Pilot Project` pattern to the design of a new API. In projects that require fast time-to-market, the API provider applies the `Play-it-fast Approach` pattern. The organization receives feedback not only from API consumers but also from end-users. These fast feedback cycles are possible due to well-established communication channels that allow API consumers to provide feedback and new ideas.

Cross-case observations:

All cases have in common that they are partner or public API initiatives. Also, they are either in production or in the pilot phase. However, the number of API consumers varies from very few to several thousand. Hence, the applicability of the pattern `Play-it-fast Approach` does not seem to depend on any of these API initiatives' characteristics.

5.5. Idea Backlog

Pattern Overview	
Name	`Idea Backlog`
Pattern Type	API Provider Internal Pattern
Summary	An idea backlog is a dynamic list that stores and aggregates consumer wishes for API endpoints derived from consumer support requests, discussions, or surveys.

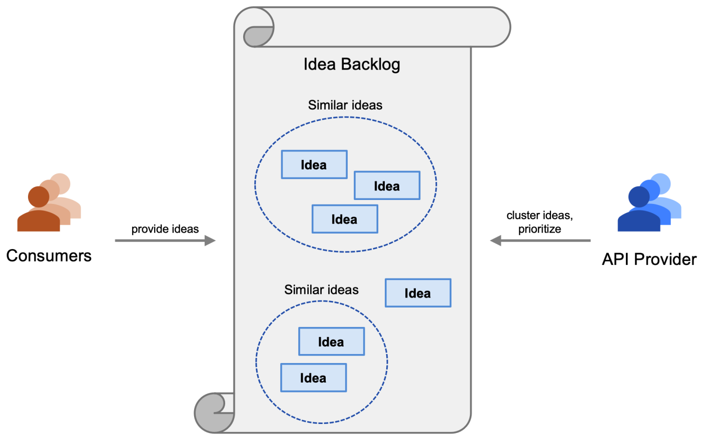

Figure 5.5.: The API provider collects and aggregates consumers' ideas for new APIs and changes to existing APIs in an `Idea Backlog`.

Context:

An API provider wants to start a new or evolve an existing API initiative. For an API initiative to succeed, the APIs must meet the API consumers' needs. However, API consumer needs can be many and very different. Thus, the API provider needs an approach to identify and integrate consumer wishes into the development process.

Concern:

How can the API provider collect and utilize information on API consumer needs?

Forces:

- *Specific vs. generic consumer needs.* API consumers request new functionality tailored to their particular needs. However, each endpoint needs to be designed and maintained by the API provider, i.e., even an API endpoint used by only one consumer results in operating costs. Hence, when designing a new or changing an existing API, the API provider has to find a balance between meeting the needs of specific consumers and making the API generic enough that the API addresses the needs of a broader range of consumers (C4).

- *Unknown consumer groups.* The API provider might not be aware of API consumer groups interested in the APIs and, thus, not of their needs.
- *Number of consumers.* For API initiatives with few consumers, the API provider can easily and informally track API consumer requests for new or changed endpoints. However, for API initiatives with many consumers, it can be difficult to track the wishes of API consumers and identify similarities between them without a structured approach (C4).
- *Process effort.* The structured collection and review of new ideas require resources.

Solution:

An idea backlog is a dynamic list that stores and organizes consumer wishes for new or changed endpoints derived from consumer support requests, discussions, or surveys. The API provider uses the idea backlog to aggregate and abstract the consumer's wishes to design new APIs or evolve existing APIs that meet consumer needs. Also, the API provider can validate planned changes against the idea backlog.

Stakeholders:

The *API provider* has to collaborate with *API consumers* to collect and verify ideas for API design. Also, *customer support* can help collect API consumer needs and wishes.

Implementation Hints:

Idea collection. First, the API provider has to collect the wishes of (potential) API consumers for new APIs or changes to APIs and enter them into the backlog. The API provider can collect such wishes through feature requests (Spichale, 2017) or abstract incoming support requests. Also, the API provider can talk to API consumers in meetings or during conferences (C4; C5). Another approach is to create and monitor online communities with end users and extract ideas from discussions (C12). Furthermore, an API provider can actively send surveys to API consumers asking for wishes (C3). Finally, the API provider could target influential developers and ask for their feedback and ideas for improving API initiatives (Jacobson et al., 2012). Such collaborations lead to improved APIs and additional marketing through the developers' channels (Jacobson et al., 2012). In all cases, the API provider should also collect context information and information on the requesting party, including contact information (C4).

Idea consolidation. In the next step, the API provider needs to organize the wishes. For example, the provider can cluster the ideas according to endpoints, use cases, or business objects (C4). This way, an API provider creates an overview of how many customers requested a feature and what variants of requests exist. In a further step, the API provider could allow API consumers to vote for ideas that are useful for them (Spichale, 2017).

Implementation. Finally, the API provider can use clustered ideas to design new APIs or evolve existing APIs to meet several API consumers' wishes (C4). The API provider must abstract the ideas in the backlog to design a generic solution that meets the wishes of as many consumers as possible (C4). The backend and provider teams implement the aggregated ideas according to their development approach, e.g., using an agile approach. In a marketplace setting, the marketplace provider can also decide not to implement a consumer wish but instead search for partners who want to realize a requested feature or product (C13).

Validation. Moreover, the API provider can validate already planned changes against the idea backlog to ensure that a consumer need for these changes exists.

Consequences:

Benefits:

- *Specific vs. generic consumer needs.* The idea backlog allows API providers to collect the specific wishes of several API consumers and aggregate them in a way that results in designing an API endpoint that meets the needs of several consumers. Hence, the API provider can design an API that might not perfectly meet the requirements of each API consumer but, in sum, balances the operating costs of the API with its overall usage across consumers (C4).
- *Unknown consumer groups.* The consumers can submit ideas for new or changed endpoints, e.g., via support requests, discussions, or surveys, and the API provider documents and analyzes these ideas. Hence, all kinds of consumers can submit ideas.
- *Number of consumers.* An idea backlog provides a structured approach for documenting and analyzing consumer requests for new or changed API endpoints, even for large numbers of API consumers.

Drawbacks:

- *Process effort.* The API provider has to design a structured process for collecting consumers' wishes in an idea backlog and allocate resources to the process execution.

Related Patterns within this Pattern Catalog:

An `Idea Backlog` helps identify relevant use cases that the provider submits to the `API Clearing Process`.

Known Uses:

We observed the pattern in five cases:

The API portal provider of an automotive organization (C3) shares vehicle data with car owners. The organization uses online surveys to collect ideas for new APIs or required changes to existing APIs to increase future adoption. In addition, they plan to create a forum that allows API consumers to exchange experiences. The API provider also wants to use the forum as a source of new ideas.

Cases C4 represents the public API initiative of a software service provider. End users with a license for the API provider's software product automatically have access to its public API to enable customized integration with their systems. The API provider collects ideas for new or changed endpoints for the public API based on support requests and personal discussions with consumers or potential consumers. All wishes are entered into an idea backlog and categorized according to business objects. The API provider team reviews the idea backlog before every quarter to define the functionality they will implement in the next quarter. During internal refinement meetings, the provider team abstracts the ideas and designs or evolves APIs specifications to be generic enough to meet the wishes of several API consumers. In the next quarter, the API providers and backend providers implement and publish the APIs using a *Play-it-fast Approach*. After publishing the initial API version, API consumers can provide feedback.

Case C12 captures a financial services provider that provides software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering an integrated solution to the end users. The organization operates an online forum that allows end users to discuss ideas for new functionality. The API provider then derives ideas from these discussions for the idea backlog.

Next, the API provider in C13 provides a core platform that allows external third-party software providers to publish modules (APIs and solutions) and end-user to consume these modules. The platform provider collects wishes from existing or potentially new consumers and reviews if there are currently providers on their platform that already provide these functionalities. If not, they approach module providers that could possibly offer these functionalities and ask them to provide these functionalities on the platform in the future. Also, in some strategic cases, the API provider decides to build new APIs themselves. However, the API provider admits that this process is currently rather manual and needs to be more structured in the future.

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[17]. As part of the Twilio developer portal, API consumers can submit feature requests[18]. In addition, other consumers or interested parties can vote for or comment on these feature requests. Hence, Twilio involves consumers in the prioritization of feature

[17]https://www.twilio.com/ (accessed 20.12.2023).

[18]https://roadmap.respond.io/feature-request(accessed 20.12.2023).

requests.

Cross-case observations:

Across all cases, we observe the pattern `Idea Backlog` in more mature API initiatives with APIs already in production. Additionally, the number of API consumers is high enough to make informal tracking of consumer wishes without a process difficult. The initiatives are primarily public, which makes sense since the pattern also allows the API provider to identify new consumer groups that the API consumer did not target in the first place. We further observe the pattern for API portals as well as marketplaces.

5.6. Testing Strategy

Pattern Overview	
Name	`Testing Strategy`
Pattern Type	API Provider Internal Pattern
Summary	A centrally defined testing strategy enforces the testing of new APIs or changes to existing APIs to reduce the likelihood of unexpected behavior of new or changed APIs or backends (C3).

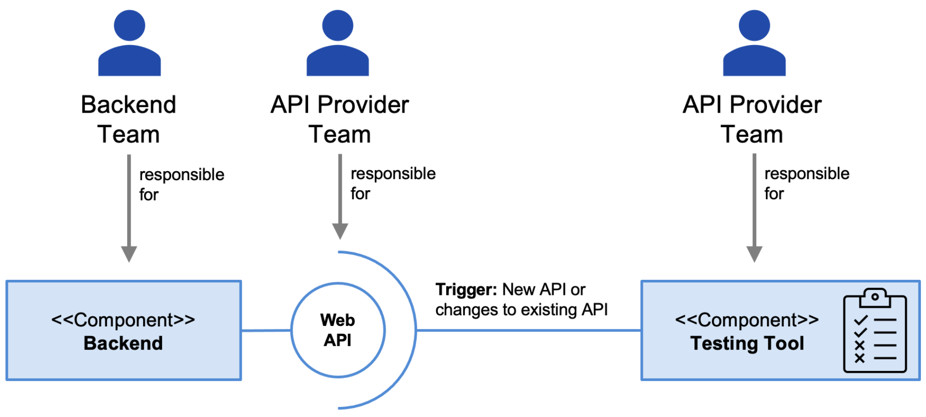

Figure 5.6.: The API provider enforces a centrally defined `Testing Strategy` for new APIs or changes to existing APIs.

Context:

API providers introduce new APIs or change APIs, e.g., to evolve their functionality or fix bugs. APIs are complex software artifacts requiring the interaction between several software components, i.e., the API itself, the backend, and potentially a gateway, usually with different teams or persons responsible for each of them. Hence, changes to one component could affect other components in unexpected ways, e.g., a backend change could impact the API's behavior unexpectedly. However, API consumers expect APIs to meet communicated functional and non-functional specifications continuously. Therefore, the API provider has to ensure that changes do not lead to unexpected API behavior, performance degradation, or any other issues impacting API consumers before releasing changes into production.

Concern:

How can the API provider ensure that new or changed APIs behave as specified regarding functional and non-functional properties?

Forces:

- *Consumer expectations.* Consumers expect APIs to meet documented functional and non-functional requirements. Failure to meet these specifications leads to frustration for API consumers, who might abandon using such an API.
- *Existing integrations.* API consumers already integrated with an API rely on the API provider not to make any breaking changes or to communicate these changes ahead of time. Unexpected changes that break a consumer integration lead to frustration and potentially to contractual fines. Hence, the API provider has to ensure that changes not meant to be breaking do not influence the behavior of the APIs.
- *Support effort.* If API consumers notice unexpected changes to an API, they will contact the support asking for a resolution. Hence, unexpected changes can lead to high volumes of support requests (C4; C5).
- *Test design.* Testing new APIs and changes to an API requires the adequate design and execution of test cases.
- *Required collaboration.* Often, different teams or persons are responsible for the different software components interacting to deliver an API. Errors arising at the interface between such components require collaboration between these teams. However, these teams might have different priorities or approaches to solving such issues.

Solution:

The API provider designs and implements a testing strategy that enforces the testing of new APIs or changes to existing APIs. The testing approach encompasses different testing types and involves all teams concerned with API provision (C3).

Stakeholders:

An *API governance* team within the *API provider* team could design and coordinate the testing strategy. Also, the API provider has to involve the *backend providers* into testing.

Implementation Hints:

Types of testing. The test strategy should comprise different types of testing conducted by various teams involved in the API initiative. These types of testing can be:

- Backend testing (C3)
- Unit testing (Medjaoui et al., 2018)
- API gateway configuration testing (De, 2017; C3)
- End-to-end testing (C3; C4; C5)

- API specification testing (De, 2017)
- Integration testing (Medjaoui et al., 2018)
- API performance and load testing (De, 2017; Medjaoui et al., 2018)
- API security testing (De, 2017; Medjaoui et al., 2018)
- API documentation testing (De, 2017)
- UX testing (Medjaoui et al., 2018)
- Production testing (Medjaoui et al., 2018)

Decentralized vs. centralized testing. Decentralized teams can execute early test stages to increase speed. However, the API provider should run later test stages in a centralized manner to ensure safety (Medjaoui et al., 2018).

Test automation. At the beginning of an API initiative with few APIs, API providers can manually test APIs. However, an increasing amount of APIs and consumers leads to high costs of manual testing (Medjaoui et al., 2018). Thus, the API provider should introduce and continuously evolve testing tools that support automated testing (Medjaoui et al., 2018; C4; C5). Additionally, the test tools potentially enable parallel testing, virtualization, and canary testing (Medjaoui et al., 2018). Also, test tools provide capabilities for test data management (De, 2017). An API provider can also enforce the testing strategy through CI pipeline configurations (C3; C14).

Test-driven development. Test-driven development and the early inclusion of testing experts in the design and development of an API usually results in improved testing solutions for the APIs (Medjaoui et al., 2018).

Consequences:

Benefits:

- *Consumer expectations.* Even though testing cannot guarantee uncovering all unexpected behaviors or errors (Medjaoui et al., 2018, p. 50), proper test design and execution can reduce the likelihood of unexpected behaviors or errors before deploying a new API or making a change (C4; C5). Thus, the new APIs or changes are less likely to affect consumer expectations negatively.
- *Existing integrations.* Testing decreases the likelihood that changes to APIs unexpectedly break API consumer integrations.
- *Support effort.* Testing decreases the likelihood of high loads of support requests resulting from unexpected behavior of changes to an API.

- *Required collaboration.* Clearly defined testing approaches include processes to handle errors arising at the interface between components, thus reducing frictions or delays in the resolution.

Drawbacks:

- *Test design.* The design and execution of testing are effortful. The API provider has to define and manage test cases and test data for functional and non-functional testing. Also, organizations potentially have to maintain test infrastructure. Nevertheless, testing cannot guarantee uncovering all unexpected behaviors (Medjaoui et al., 2018).

Related Patterns within this Pattern Catalog:

The provider should apply a `Testing Strategy` for testing a new API that the provider introduces through a `Collaborative Pilot Project` or `Play-it-fast Approach`.

When testing new or changed APIs, the provider creates test cases as part of the `Testing Strategy`. These test cases can be reused for `API Quality Monitoring`.

In addition, a `Service-Level Agreement (SLA)` defines the thresholds for non-functional requirements of an API. The `Testing Strategy` must ensure that APIs meet these thresholds before publication.

Other Related Patterns:

The book Richardson (2019) together with the online resource Richardson (n.d.) present the pattern `Consumer-driven Contract Test`, which captures the idea of a consumer writing a test suite for an API to ensure that it meets the consumer's needs. Also, `Service Component Test` (Richardson, n.d., 2019) describes the testing of services in isolation. An API provider can use `Consumer-driven Contract Tests` and `Service Component Tests` as part of the `Testing Strategy`.

Furthermore, Fowler (2003) presents the `Service Stub` pattern. A service stub is a locally running, fast service that replaces a service outside a development team's control for easy testing. API providers can use service stubs in the context of a `Testing Strategy`.

Known Uses:

We observed the pattern in four cases:

The API portal provider of an automotive organization (C3) shares vehicle data with car owners and workshops and anonymized road condition data with government bodies. New APIs and changes to existing APIs have to pass backend testing, gateway testing, and end-to-end testing before being released into production.

Cases C4 and C5 represent a software service provider's public and partner API initiatives. End users with a license for the API provider's software product automatically have access to the public API to enable customized system integration. The partner API allows other software providers to create integrations with their software products, thus managing more integrated solutions for end-users. The API gateway team is responsible for the API overall testing strategy in these two cases. The API gateway team uses a commercial testing framework to create and automate end-to-end tests, ensuring that the APIs behave as expected by the consumers. Successful testing is part of the Definition of Done (DoD). Hence, the backend and API providers only deploy new APIs or changes if all tests are successful. This testing strategy is supported by the DevOps culture of the organization, meaning that developers are interested in publishing high-quality software since they have to maintain and evolve it afterward.

Finally, case C12 captures a financial services provider that provides software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering an integrated solution to the end users. The organization enforces its testing strategy through its CI pipeline configuration.

Cross-case observations:

These cases have little in common. The API initiatives are private or public, have different amounts of API consumers, and are partially in a pilot phase or already in production. This heterogeneity in API initiative characteristics makes sense since software testing is a universally recognized approach to ensure software quality in different contexts and plays an essential role in modern development practices, including agile development and DevOps.

5.7. API Clearing Process

Pattern Overview	
Name	`API Clearing Process`
Pattern Type	API Provider Internal Pattern
Summary	A data clearing process ensures that all API endpoints comply with legal and strategic requirements before they are published externally by involving different stakeholders who provide feedback and need to sign off on a new API or the change to an existing API.

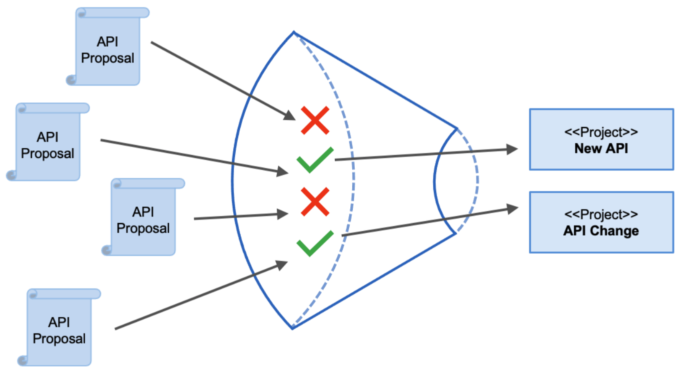

Figure 5.7.: All new APIs or major changes need to pass the *API Clearing Process*.

Context:

An employee or team belonging to the API provider has the idea to create a new API or perceives the need for a significant change to an existing API. The employee or team assumes a market demand for the new or changed API. However, introducing a new API or changes to an existing API creates effort and is connected with risks, e.g., legal or security risks.

Concern:

How can an API provider ensure that new or changed APIs create profit for or at least don't harm the API provider's organization?

Forces:

- *Return on investment.* The design and implementation of new APIs or changes to existing APIs are costly (C3; C6). Usually, creating a reusable API is more effort compared to designing and implementing a point-to-point integration. The additional effort is caused, e.g., by the need to design the API in a generalist way that allows for reuse and the need to design monetization models (C6). Therefore, a solution should ensure that the additional effort of creating and changing APIs yields a positive return on investment (C3).

- *Risks.* Publishing an API to external users is accompanied by risks. For example, the API provider could accidentally publish privacy-relevant or strategic data, or open up the application landscape to security attacks.
- *Play it fast.* If new or changed APIs enable new business opportunities for API consumers, API providers need to introduce the APIs into the market quickly. Competing API providers might try to offer similar data or functionality if they perceive a demand in the market (C3).
- *Centralized decision making cost.* Centralizing decisions creates cost, since the organization needs funds to maintain the governance body, but also in the sense of potential decreases in employee happiness and rate of technological innovation (Medjaoui et al., 2018).

Solution:

The API provider designs and coordinates a data clearing process to ensure that any API endpoint meets legal, strategic, and other requirements before being exposed externally. An employee or team that wants to create a new API or make a significant change to an existing API has to prepare specifications and a business case and present it to stakeholders from different departments. The stakeholders assess the proposal and potentially request improvements. The API provider can implement the API only after successfully passing the data clearing process.

Stakeholders:

The *API provider* involves the *legal*, *finance & controlling*, *sales & marketing*, and *research & development* departments into the review of new APIs or major changes to APIs.

Implementation Hints:

Process goal. The *API governance* or another central entity designs and manages the data clearing process. Hence, the pattern `API Clearing Process` describes a process for centralized decision making. An API provider should centralize decisions if they can impact a system negatively and in an irreversible way (Medjaoui et al., 2018). For example, governance should centralize API security since a vulnerability can lead to extensive and long-lasting damage (Medjaoui et al., 2018). Overall, the purpose of the data clearing process is to ensure that all endpoints are assessed for legal, strategic, and other requirements to increase the likelihood that the implementation of a new API or changes of an existing API leads to positive results for the API provider (C3). In case of changes to an existing API, the process is only relevant for major changes or customer-facing changes (C11).

Process steps. Ideas for new APIs or changes to existing APIs emerge from new or changing business agreements or experiences of community managers, business analysts, or solution architects (De, 2017). An employee or team with an idea for a new API or suggestions for changes of an existing API start the data clearing process. The API provider should define requirements for the initiation and successful execution of the data clearing process (see below). Moreover, the team organizing the data clearing process can make available best practices supporting and preparing the employees or teams (C6).

Once the process starts, the organizing unit identifies all stakeholders potentially affected by the new API and departments that can support data clearing with their expert knowledge. The involved stakeholders potentially comprise *API provider* teams, business departments, *legal*, *finance & controlling*, *sales & marketing* and others (C3).

In the next step, all stakeholders review the idea for a new API. Each stakeholder can reject or propose changes to the propsal for strategic, financial, privacy-related, or other reasons. Besides not meeting any of the data clearing process's requirements, reasons to reject a new API can be that publishing the data or functionality would disrupt the business or bad past experiences with the potential API consumer. Potentially, the employee or team pitching the new API has to improve the idea iteratively until it meets the demands of all stakeholders of the data clearing process.

The data clearing process only approves the new API if it satisfies the demands of all involved parties. Afterward, the API provider can initiate the implementation of the API (C3).

Requirements. The data clearing process can place various requirements on an API:

- The prescription that each API needs to have a strategic goal (Medjaoui et al., 2018).
- The calculation of a business case that considers potential partners that already showed interest in the idea, expected sales, acquired funding (Medjaoui et al., 2018), and migration costs. Potentially, *finance & controlling* can support the creation of the business case (C3).
- A partnership with at least one API consumer who commits to participating in a pilot project or provides continuous feedback during API implementation according to the pattern `Collaborative Pilot Project` (C3).
- The creation of a technical specification that complies with the organization's guidelines, e.g., regarding documentation, reviews, versioning approach, and technology choices (De, 2017). This step should also entail an analysis of the impact of the proposal on existing functionality.
- The creation of a prototype to show the feasibility (C3) and the validation of the prototype from the consumer perspective (Medjaoui et al., 2018), to show the APIs value.

- The appointment of the future `API Product Owner` for the API (C3).
- The successful passing of legal and privacy assessments (Jacobson et al., 2012). The future API provider usually conducts these assessments in close collaboration with the *legal* department (C11).
- The compliance with an organization's security guidelines (Medjaoui et al., 2018; C6).
- The removal of any internal terminology or knowledge attached to the endpoint (C3).

These exemplary requirements are not exhaustive, and the API provider needs to adapt them to an organization's needs.

Consequences:

Benefits:

- *Return on investment.* A data clearing process can entail the calculation of a business case in cooperation with experts from the *finance & controlling* department. Furthermore, the collection of feedback and execution of pilot projects with potential consumers validates market demand. Security and legal assessments ensure that the API provider does not have to shut down a new or changed API after implementation due to security and legal issues. Thus, the data clearing process lowers the likelihood of a loss of investments (Medjaoui et al., 2018; C3).
- *Risks.* The thorough vetting of new APIs or major changes to APIs reduces the risks of harming the API provider organization, e.g., by accidentally opening up systems to security vulnerabilities.

Drawbacks:

- *Play it fast.* A data clearing process can eat up a lot of time, especially if the process requires the involvement of many different stakeholders. For example, resource limitations within the governance body might decrease decision throughput and thus decrease time-to-market (Medjaoui et al., 2018). Thus, the API provider might introduce new APIs and changes to existing APIs late (C3).
- *Centralized decision making cost.* The centralization of decisions using a data clearing process creates cost for maintaining the governance body. In addition, employee happiness and the rate of technological innovation might decrease (Medjaoui et al., 2018).

Related Patterns within this Pattern Catalog:

The API provider derives ideas for new APIs or changes to existing APIs from the `Idea Backlog`. As soon as the new API or changes pass the `API Clearing Process`, they are realized in a `Collaborative Pilot Project` or using a `Play-it-fast Approach`.

Moreover, the provider could make it a requirement for APIs to appoint a `API Product Owner` to pass the `API Clearing Process`.

Known Uses:

We observed the pattern in four cases:

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops. New APIs have to go through a data clearing process. The person steering the idea of a new API through the data clearing process should also be the future product owner of the API. The data clearing process of the organization involves IT, legal, finance & controlling, aftersales, sales & marketing, research & development, and other affected departments. The data clearing process requires at least one external partner that commits to participate in a `Collaborative Pilot Project` before approving an API. The API provider uses a simplified data clearing process for legally prescribed APIs. The goal of the data clearing process is to prevent unsuccessful investments.

The partner API portal of a software and IT service provider (C4) integrates software products of other ecosystem players with the API portal providers software to present end users with more integrated overall solutions. An API governance reviews APIs only after their implementation.

Case C6 comprises an API initiative within a group setting. The API provider organization is a subsidiary that manages the API platform and supports other subsidiaries with the API design, provision, and consumption. A central entity within the API provider organization assesses each API that a subsidiary wants to expose to ensure compliance with existing guidelines, e.g., security guidelines.

The API provider of C10 offers a financial service for API consumers to integrate into their systems. The `API Product Owner` has to get legal to approve the introduction of new APIs or major consumer-facing changes.

Cross-case observations:

The cases do not have much in common. They span public, partner, and group API initiatives and have few or high numbers of users. Also, the types of clients range from businesses to governmental institutions and individual developers. Also, monetization differs between the API initiatives, including contractual agreements, per-call fees, and non-monetized APIs. However, what all API initiatives have in common, is that they are already in production. Also, we observed the pattern in rather big organizations operating in traditional industry sectors, which makes sense since these organizations are often more prone to processes.

5.8. API Facade

Pattern Overview	
Name	`API Facade`
Pattern Type	API Provider Internal Patterns
Summary	An API facade abstracts the invocation of several backend services into a single API (Gamma et al., 1994). The API facade thereby supports the tailoring of APIs that fit the user stories of the API consumers.

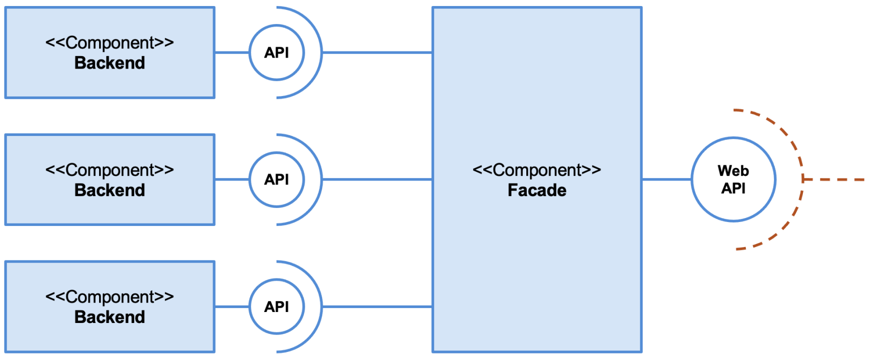

Figure 5.8.: The `API Facade` presents a unified interface to a set of backends (Gamma et al., 1994).

Context:

An API provider wants to realize a use case for an API consumer but must draw on the data or functionality of several backends (C4). For each backend, an internal API already exists. However, the internal APIs are structured according to internal needs and afflicted with internal knowledge, which the provider wants to hide from consumers (C3).

Concern:

How can an API provider create an API tailored to external API consumers' needs that draws on several internal backend APIs?

Forces:

- *Developer friendliness.* An API consumer wants to integrate with a developer-friendly, high-quality API (C2). Hence, an API consumer prefers a single API that realizes their use case over a set of APIs for which they must first implement some logic.

- *Waste.* The implementation of a use case might only require access to a subset of functionality and data of an internal API. A solution should prevent an API consumer from having to fetch unnecessary data.
- *Sensitive data and functionality.* Implementing a use case might require access to an internal API that exposes sensitive functionality or data. The API provider might need to hide some data for legal or strategic reasons (C3).
- *Flexibility.* The backend systems evolve. The API provider should reduce the impact of changes on client applications to a minimum.
- *Internal API initiatives.* Some organizations have internal API initiatives, meaning that either the upper management or a central governance body motivates or forces development teams to create APIs for internal data exchange.
- *Coordination effort.* The solution should require as little coordination effort between teams as possible (C4).
- *Issue tracing.* A solution should ensure easy issue tracing across all associated components involved in realizing a use case (C3).
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.

Solution:

The API provider can introduce an additional software component as a `API Facade` (Erl, 2008; Gamma et al., 1994) between internal backend APIs and external API consumers (De, 2017). The facade allows the API provider to logically link functionality and data of different backends and design a single, simple API that meets the consumers' needs (Gamma et al., 1994). The exposed API can also hide technology choices, remove internal knowledge, and filter data (De, 2017).

Variants:

The API gateway can be a facade shielding the backends (De, 2017). Using an API gateway as a facade is intuitive since it is usually the single entry point into an organization's systems already (De, 2017). The API gateway as a facade usually realizes mainly traffic management tasks, including security enforcement, data validation, message transformation, traffic throttling, and routing (De, 2017; Jacobson et al., 2012). Therefore, an API gateway can help API development teams to focus on API development by relieving them from infrastructure tasks (Jacobson et al., 2012). However, an API gateway could also realize the composition of backend and database services (De, 2017; Jacobson et al., 2012).

Stakeholders:

The *API provider* has to collaborate with several *backend providers* to access and understand their APIs. Furthermore, the API provider should communicate with the *API consumer* to ensure that a new API meets their needs.

Implementation Hints:

Goals of the facade. An `API Facade` is an additional component between internal backend APIs and external API consumers (Erl, 2008; C2). The API provider should design the layer to logically orchestrate the functionality and data of several backends in a way that realizes new use cases for API consumers (De, 2017; Erl, 2008; Spichale, 2017; C2; C3; C4; C6). The logic implemented in the `API Facade` can be very complex, e.g., realizing several sequential or parallel calls to the backend, or relatively simple, e.g., renaming endpoint fields (C3; C6). The goal of the `API Facade` is to improve the usability of the API (Gamma et al., 1994; C2; C6; C12). Another goal of the `API Facade` should be to hide technology choices of the backend (De, 2017; C6). Finally, the `API Facade` layer should hide internal knowledge linked to internal APIs (C3; C12).

Technology choice. An API provider can choose different technologies to implement the `API Facade`. An option that is gaining popularity with digital organizations is GraphQL[19]. GraphQL allows, for example, targeted data queries without waste, easier versioning, and safe deprecation of unused fields (C10).

Consequences:

Benefits:

- *Developer friendliness.* A facade allows the API provider to design a new API tailored to the use cases of the API consumers (C2). Integration efforts for consumers decrease since a client application communicates with only one API instead of several backend APIs (De, 2017). Hence, a facade reduces chattiness between backend and client application since the client application has to make fewer calls (De, 2017). In addition, the facade can centralize security negotiations, meaning that the API consumer only authenticates with the facade once, instead of having to authenticate with each backend system (De, 2017). Hence, the resulting API is more developer-friendly (De, 2017).
- *Waste.* A facade can exclusively expose the functionality and data of a backend API needed to realize a use case. Thus, the API consumer does not over-fetch data.
- *Sensitive data and functionality.* A facade can expose only the functionality and data of a backend API needed to realize a use case. Therefore, sensitive functionality and data remain hidden (C3).

[19]https://graphql.org/ (accessed 20.12.2023)

- *Flexibility.* A facade increases the flexibility and agility of a system architecture by allowing the API provider to replace certain backends without impacting client applications (De, 2017; Erl, 2008; Gamma et al., 1994; Spichale, 2017).
- *Internal API initiatives.* A facade can draw on existing internal backend APIs and orchestrate them into a use case (C3).

Drawbacks:

- *Developer friendliness.* The design of a developer-friendly, high-quality API is difficult. A facade provides some design flexibility, but the API provider still needs to put effort into discovering a good design. Also, it is more difficult to create a good design if you realize a more complex use case drawing on several backend APIs instead of exposing a simple functionality (C2).
- *Internal API initiatives.* Not all teams put the same effort into realizing internal API initiatives. Thus, APIs can be of bad quality, and the API provider either needs much effort to enhance them with a facade or can not use them at all (C3).
- *Coordination effort.* The realization of a use case with an API facade requires access to different backend APIs, and therefore the teams responsible for these backends must collaborate. Such collaboration can be time-consuming. Also, teams might have different priorities, leading to conflicts (C4).
- *Issue tracing.* A facade can make issue tracing more difficult. If an error occurs, a facade with issue tracing capabilities can identify issues within the facade component or point to the backend producing the issue. However, some issues also result from dependencies between backends. These issues are often difficult to identify and complex to fix (C3).
- *Effort.* The introduction of an additional component introduces design effort, development effort, and communication effort (Erl, 2008).

Related Patterns within this Pattern Catalog:

A `API Facade` can be used to implement a public `Web API`.

Other Related Patterns:

An `API Facade` is a specialized implementation of the `Facade` pattern presented by Gamma et al. (1994) and Buschmann et al. (2007a). In Fowler (2003), the same pattern is presented under the name `Remote Facade` and Erl (2008) names the pattern `Service Façade`. In contrast to a `Proxy` or an `Adapter` that capsules the interface of one component, a `Facade` capsules a whole subsystem and presents a new, simplified API (Gamma et al., 1994; Spichale, 2017).

Known Uses:

We observed the pattern in seven cases:

The API initiative of C2 makes a simulation and modeling algorithm accessible. The API provider implemented an adapter and a REST API between the backend components and the API consumers. The goal of the facade layer is to ensure a clear structure and good performance.

The API portal provider of an automotive organization (C3) makes several APIs available to car owners and businesses. Internally, a guideline animates developer teams to create APIs for data exchange between departments as part of every new project. However, the API design is an additional effort, and developer teams do not always design the APIs with high-quality structures. Therefore, the API provider creates a facade layer for internal APIs before they are exposed externally. Besides improving the API structure, the facades also connect data and functionality of several backends, remove sensitive data, and hide internal knowledge. The only APIs used internally and externally without an orchestration layer are APIs built for regulatory compliance with precise structure specifications.

The public API portal of software and IT service provider C4 allows other ecosystem players to integrate their software products with the API portal provider's software to present end users with more integrated overall solutions. The same organization has a separate partner API initiative C5, allowing users of their software product to integrate it with the consumers' system landscape. Both API initiatives draw on backend functionality and data that several functional backend development teams implement and maintain. However, the API provider team does not have a functional area but implements an API layer on top of the functionality and data of the backends. During regular coordination meetings, the API provider and backend teams discuss the resulting dependencies.

Case C6 describes an API initiative within a group setting. The API provider organization is a subsidiary that manages the API platform and supports other subsidiaries with the API design, provision, and consumption. The API provider creates an API utilizing the backends of other subsidiaries. These `API Facades` hide technologies, augment the API with the functionality of several systems or cut the API differently. Depending on the use case, a facade layer can implement a lot of logic or be simple. The API provider created a guideline for backend providers and API provider teams that guides the creation of such services.

The API initiative C10 provides financial services to partners. The partners are high-volume consumers with special requirements for their API. All backend development teams create APIs and register them in a central registry. Thus, the API provider team serving an API consumer implements these special requirements in a facade that uses the backend APIs.

The same organization also offers a pre-defined financial service API (C11) for low-volume API consumers to integrate into their systems. The API provider does not create a customized API for each API consumer, but the consumers must use the existing interface. The organization uses a GraphQL orchestration layer for the financial service interface to call the REST APIs of the different backends that make up the financial service. However, the orchestration layer is not visible to the API consumer. Instead, the API consumers interact with a JavaScript SDK. A dedicated team develops and maintains the orchestration layer. The organization plans to expose the GraphQL API directly in the future.

Cross-case observations:

The cases are heterogeneous, spanning public, partner, and group API initiatives. Also, the number of consumers ranges from small to very high amounts, and the consumers are primarily businesses but also government intuitions and single developers. Furthermore, the monetization approaches differ and comprise contractual, pay-per-use, free use as part of a software license, and completely free schemes. However, all cases are in production except for one API initiative in a pilot phase. Also, all API initiatives are developer portals and not marketplaces.

5.9. API Quality Monitoring

Pattern Overview	
Name	`API Quality Monitoring`
Alias	Error reporting (Medjaoui et al., 2018), Service-level monitoring (De, 2017)
Pattern Type	API Provider Internal Pattern
Summary	API quality monitoring describes continuously testing an API's non-functional properties to detect anomalies and take countermeasures quickly.

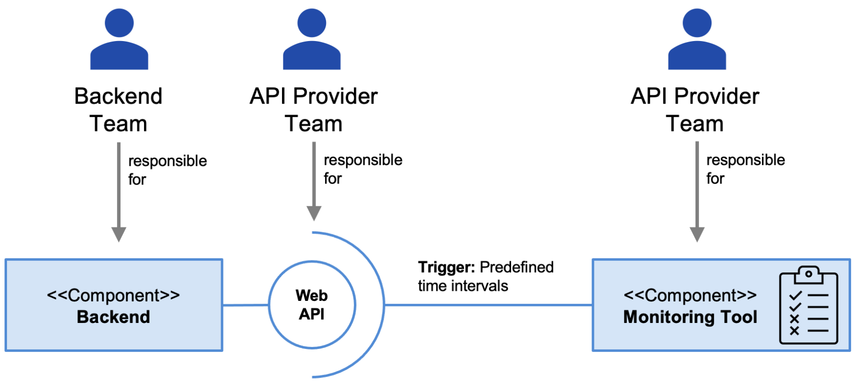

Figure 5.9.: According to the pattern `API Quality Monitoring`, the provider continuously monitors the non-functional properties of an API in production.

Context:

API consumers integrating APIs to support a business model rely on the APIs' performance, e.g., low latency and high availability. Therefore, API consumers might only adopt an API if the API provider guarantees specific non-functional API properties in a `Service-Level Agreement (SLA)`. However, even if the API provider has a `Testing Strategy` in place, API testing cannot guarantee to uncover all unexpected behaviors in case of changes to an API (Medjaoui et al., 2018). Also, new issues might arise during operations. Hence, an API provider should be aware of anomalies or degradation of an API's performance to react to such issues quickly.

Concern:

How can an API provider identify performance anomalies or degradation of APIs in production in a timely manner?

Forces:

- *Awareness and resolution of issues.* API consumers integrate APIs to support their business models only if they can rely on their performance, e.g., high availability and low latency. If APIs do not meet the API consumers' requirements, they will not adopt them (Jacobson et al., 2012) or abandon them (De, 2017). Also, API consumers might only adopt an API if performance guarantees are in place, i.e., an `Service-Level Agreement (SLA)`. Such SLAs can tie failures to meet the guaranteed performance to financial fines. Hence, API providers must be aware of and resolve performance issues as quickly as possible.
- *Error-free operations.* Testing of APIs cannot guarantee to uncover all unexpected behaviors (Medjaoui et al., 2018) and new issues might arise during operations. Such issues should be fixed before the API consumers become aware of them.
- *Support effort.* As soon as consumers notice an API performance degradation, they contact the support. Therefore, unexpected performance issues lead to high support requests and many support employees trying to solve the issue in parallel.
- *Monitoring design.* Proper monitoring requires the API provider to design or buy, implement or configure, and operate a monitoring system.

Solution:

API quality monitoring describes the continuous testing of an API's non-functional properties in production. API quality monitoring allows API providers to detect anomalies and take countermeasures quickly.

Variants:

In a marketplace setting, the marketplace provider is a neutral instance acting as a broker between an API provider and an API consumer. In such settings, the marketplace provider should monitor the performance of the APIs to create neutral reports and alert both the API provider and the API consumer.

Stakeholders:

The *API provider* has to collaborate with the *backend provider* in case an anomaly is rooted in the backend implementation.

Implementation Hints:

Measurable requirements. A prerequisite for API quality monitoring is the definition of measurable performance KPIs. An example of such a requirement is the achievement of 99,99% availability (C10; C11). The API provider should derive such KPIs from the organizations and

API initiatives strategy (Medjaoui et al., 2018). Furthermore, API consumers can request the definition of performance KPIs as part of an `Service-Level Agreement (SLA)`.

Test cases and data. Given the requirements, the API provider has to create test cases and data. If possible, the API provider can reuse existing test cases and data, e.g., generated during testing new APIs, changes to existing APIs, or changes to backend systems according to the `Testing Strategy`. The test cases should be end-to-end tests of the API in production, meaning that they test the behavior of the API from a consumer perspective (C3).

Tooling. At the beginning of an API initiative, a monitoring system can be simple, but with a growing API landscape, the API provider should adopt a more sophisticated monitoring strategy and standardize the monitoring approach (Medjaoui et al., 2018). Monitoring requires gathering and analyzing large amounts of data which can become challenging. Hence the API provider should buy or design suitable tools to realize a monitoring strategy (Medjaoui et al., 2018). The choice and configuration of a monitoring tool that enables monitoring automation is a significant decision (C3). The monitoring tool executes the tests and logs the results of the tests. Often, API management platforms also provide the capabilities for API traffic logging and analytics (De, 2017; C14).

Dashboards and Reports. Monitoring comprises gathering data on the APIs' performance and health and transforming it into meaningful reports or dashboards (Medjaoui et al., 2018). Usually, monitoring tools support the creation of such dashboards (C4; C5).

Automated internal notifications. Monitoring tools should automatically notify the API provider and the responsible backend provider team in case of detected anomalies (Jacobson et al., 2012; C3; C13). These notifications can be human-readable, but generating machine-readable reports might also make sense. For example, specific machine-readable notifications could automatically trigger, e.g., the allocation of additional backend resources or instant redeployment (Stocker et al., 2018).

Automated notification of consumers. API providers should proactively inform API consumers about performance disruptions (Jacobson et al., 2012) and make statistics on API performance available to API consumers (De, 2017), e.g., via the API provider portal or social media (Jacobson et al., 2012). Furthermore, the API provider should inform the API consumer about known performance disruptions for specific APIs in the ticket form when the developer opens a ticket for the respective API according to pattern candidate `Contact Form Automation`. Such a measure could reduce the number of repetitive tickets (C3).

Monitoring impact. The API provider should employ mechanisms that ensure that logging does not impact the performance of the API (De, 2017). Such a mechanism is the use of a `health check endpoint` (Richardson, n.d., 2019).

Action plans. The knowledge of a performance issue is only helpful if processes are in place to resolve these issues. Hence, the API provider needs action and escalation plans to react to these alerts (Jacobson et al., 2012).

Data volumes. Monitoring requires gathering and analyzing extensive amounts of data (Medjaoui et al., 2018). A possible approach to managing vast amounts of monitoring data is that distributed API provider teams collect data for their respective API and only pass a subset of the data to a central storage (Medjaoui et al., 2018). Then, a central API team uses the data in the central storage to derive information across the whole API initiative to reveal patterns across endpoints (Medjaoui et al., 2018).

Consequences:

Benefits:

- *Awareness and resolution of issues.* API quality monitoring enables API consumers to identify performance issues quickly (Medjaoui et al., 2018). Hence, monitoring increases the likelihood that consumers are satisfied with an API's performance and prevents SLA breaches and their consequences.
- *Error-free operations.* Monitoring enables the API provider to identify performance issues that testing did not capture.
- *Support effort.* The API provider can communicate a performance issue on the developer portal, thus informing API consumers that they do not need to open a ticket. Also, the API provider can inform the support team about known issues and ongoing resolution activities, which enables them to use their resources efficiently.

Drawbacks:

- *Awareness and resolution of issues.* Even if API quality monitoring enables API consumers to identify performance issues quickly, the API provider still has to resolve the issues. Hence, monitoring is only effective if the API provider has well-defined processes in place to address identified issues quickly.
- *Error-free operations.* Even if monitoring enables the API provider to identify performance issues that testing did not capture quickly, it is very likely that consumers notice these issues as well (C3; C4; C5).
- *Monitoring effort.* A monitoring system's design, implementation, and operation are effortful. The monitoring system needs to collect, store, and analyze potentially big amounts of data (Medjaoui et al., 2018).

Related Patterns within this Pattern Catalog:

First, `Service-Level Agreement (SLA)`s define performance thresholds for `Web APIs`, `Frontend Ventures`, or a `Client Library`. `API Quality Monitoring` enables the API provider to identify breaches of these thresholds (Jacobson et al., 2012; C4; C5).

Also, the provider designs and implements test cases and data as part of the `Testing Strategy` that can be reused for `API Quality Monitoring`.

Other Related Patterns:

The book Richardson (2019) and the online resource Richardson (n.d.) present a pattern language for microservices. Given a microservices setting, some of these patterns refine the pattern `API Quality Monitoring`. First, the pattern `Health Check API` prescribes the design of a health check API endpoint, i.e., an endpoint with the sole purpose of returning data on the health of a service. A monitoring service invokes the endpoint periodically to retrieve the service's health data. In addition, the pattern `Application Metrics` captures the implementation of a service that gathers statistics about operations, creates reports, and alerts developers.

Similarly, Hohpe and Woolf (2003) describe the `Test Message` pattern. The pattern covers the core idea of regularly sending messages to a component and analyzing the component's response for testing purposes.

In comparison, Rotem-Gal-Oz (2012) presents the `Service Watchdog` pattern. The pattern is broadly formulated and captures the active monitoring of a service's internal state and the implementation of self-healing approaches. Also, it entails the continuous publication of a service's status. Hence, the `Service Watchdog` pattern comprises `API Quality Monitoring`.

Finally, Dyson and Longshaw (2004) presents the patterns `Continual Status Reporting` and `Operational Monitoring and Alerting`. While `Continual Status Reporting` describes creating a reporting interface or protocol that displays and logs the status of relevant operational system components. `Operational Monitoring and Alerting` captures the practice of identifying indications of failures, thus enabling preventive actions. These patterns can support the implementation of `API Quality Monitoring`.

Known Uses:

We observed the pattern in nine cases:

Cases C4 and C5 represent a software service provider's public and partner API initiatives. End users with a license for the API provider's software product automatically have access to the public API to enable customized system integration. The partner API allows other software providers to create integrations with their software products, thus managing more integrated solutions for end-users. The API provider applies the pattern `Testing Strategy` for both API initiatives to prevent issues in the production environment. Nevertheless, sometimes testing misses issues, and errors arise in production. Monitoring of API response success rates alerts the API provider in case of such issues. However, the API provider states that the API consumers also almost always notice these issues.

The API provider of C10 and C11 offers a public and several partner APIs for easy integration of a financial service into websites and online shops. This organization built an analytics tool in-house. The analytics tool provides dashboards to API provider teams showing statistics on the number of API calls over time and the availability. Across the organization, all API provider teams strive for a 99,999% availability. In case the availability drops below this threshold, the analytics tool alerts the API provider team.

Case C12 captures a financial services provider that provides SaaS software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering standard integrations to the end users. The organization uses the out-of-the-box monitoring capabilities of the commercial API gateway portal in use.

The case C13 applies the pattern `API Quality Monitoring` as described in the variant. The case describes an API marketplace, i.e., the organization provides a core platform that allows external third-party software providers to publish modules (APIs and solutions) and end-user to consume these modules.

In the past, if a module consumer encountered issues with a third-party module, the consumer and the third-party software provider often created separate monitoring reports. Unfortunately, the individual monitoring reports created by different parties often deviated, leading to confusion and friction between the involved parties. In subsequent escalation calls, discussions often focused on which report was right and which party to blame instead of finding an actual solution. Thus, the marketplace provider implemented a central monitoring system that monitors the third-party software providers' modules' health status. In case of anomalies, the API marketplace provider automatically alerts all involved parties and makes the neutral monitoring report available as a basis for problem resolution.

Furthermore, the marketplace provider organization of case C13 sees itself as responsible for ensuring the quality of offered third-party modules. A major selling point of the marketplace is that all offered third-party modules and their providers have been thoroughly vetted. The monitoring system enables the marketplace to check each module's performance continuously. If a module does not provide the expected performance, the marketplace requests the module provider to take measures to improve the performance or the module is removed from the marketplace.

Stripe (C16) provides APIs for online payment processing[20]. Stripe monitors their APIs' performance and proactively communicate performance degradation or outages to API consumers on their API developer portal. Stripe visualizes API uptimes over the last 90 days on their developer portal and publishes outage updates on Twitter[21].

Twilio's (C17) is an API provider for customer engagement using voice, messaging, video, and email services[22]. It presents an overview of currently open incidents and the API response times over time[23]. In addition, the Twilio developer portal provides information about scheduled maintenance and past incidents and enables developers to contact the support [24]. Finally, interested parties can sign up for updates on created, updated, or

[20]https://stripe.com/en-de/about (accessed 20.12.2023).
[21]https://status.stripe.com/ (accessed on 20.12.2023).
[22]https://www.twilio.com/ (accessed 20.12.2023).
[23]https://status.twilio.com/ (accessed on 20.12.2023).
[24]https://status.twilio.com/ (accessed on 20.12.2023).

resolved incidents which they can request via different channels, e.g., email, SMS, webhook, or Twitter[25].

Cross-case observations:

We observe that all API initiatives applying the pattern `API Quality Monitoring` are in production, which makes sense since monitoring is only relevant for API initiatives in production. Furthermore, most API initiatives in the case base have in common that they employ commercial gateway systems which provide quality monitoring capabilities.

However, concerning other characteristics, the cases differ. For example, the cases comprise public and partner API initiatives, developer portals and marketplaces, different amounts of users, and different volumes of API products. Thus, `API Quality Monitoring` is a pattern suitable for a broad range of API initiatives.

[25]https://status.twilio.com/# (accessed on 20.12.2023).

CHAPTER 6

API Consumer-facing Patterns

This chapter presents ten *API Consumer-facing Patterns*. API Consumer-facing Patterns comprise patterns concerned with the interaction between the API provider and an external consumer.

6.1. Role-based Marketing

A previous version of this pattern has been published in Bondel et al. (2022). In this pattern catalog, we evolved the pattern.

Pattern Overview	
Name	`Role-based Marketing`
Pattern Type	API Consumer-facing Pattern
Summary	Role-based marketing denotes the clear separation of marketing material and other consumer-facing resources targeted at different user roles in the developer portal.

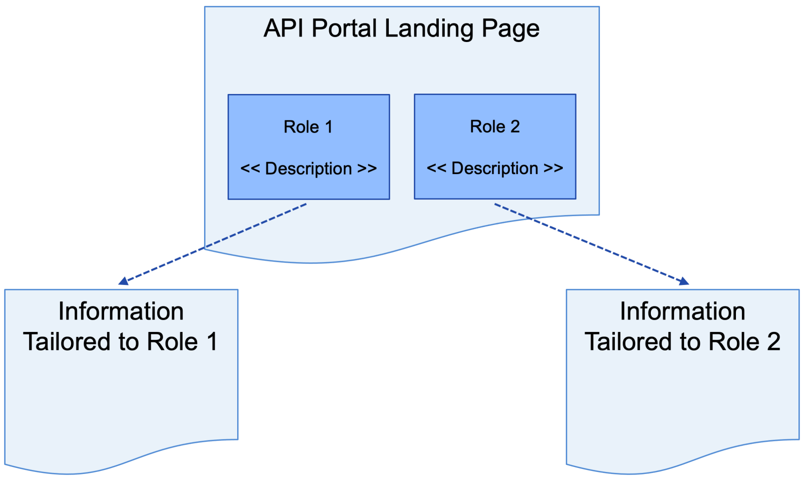

Figure 6.1.: According to the pattern `Role-based Marketing`, the API provider presents information tailored to different roles on an API portal landing page.

Context:

API initiatives can target different types of user roles. For example, in established organizations, technical and non-technical stakeholders are involved in buying an API (C3; C13). Similarly, in marketplace settings, the API provider has to address the information needs of platform users (API consumers) and third-party developers (API providers) (C13).

Concern:

How can an API portal provider address the information needs of different API stakeholder groups?

Forces:

- *Business information.* The API provider offers an API of strategic relevance for the API consumers' business model. Integrating a strategically relevant API has far-reaching consequences since the consumers' business relies on the API. Thus, business stakeholders like business owners or *upper management* want to evaluate the API from a business point of view (De, 2017) and be involved in the buy decision. The procurement of a consumer organization can be another non-technical stakeholder involved in a buying decision (C13). Hence, the business stakeholders need to understand an API but are often overwhelmed by technical specifications (C3; C13).
- *Technical information.* The API provider has to convince developers of the technical capabilities of an API. Developers need technical specifications to understand and work with an API. They do not want the API provider to flood them with business-related marketing information (Spichale, 2017; C3).
- *Consumer types.* A marketplace provider has two stakeholder groups which are third-party providers offering modules or APIs on the platform and users or API consumers that consume these modules. Both stakeholder groups have to understand the marketplace platform and its benefits. These two roles have different goals and often also different IT capabilities. While third-party developers are usually tech-savvy, the main business of platform users is not always technical. Therefore, they also have different information needs (C13).
- *Effort.* The API provider has limited resources. Thus, the solution should create little additional effort for the API provider organization.

Solution:

Role-based marketing denotes the design, maintenance, and clear separation of marketing material and other consumer-facing resources in the developer portal targeted at different stakeholder roles. The API provider tailors the resources to the information needs of the respective stakeholders, potentially in collaboration with the *sales & marketing* department. The different types of resources have to be clearly distinguished and easily navigable to ensure that additional information does not interfere with the other stakeholders's user journey.

Variants:

We observed the pattern in two types of settings. First, in settings with an API provider that offers an API via an API portal to partners or the public, the API consumers often involve developers and business stakeholders in the decision to use an API. Hence, the API provider has to address the technical information needs of developers and the business information needs of the business stakeholders in these settings (C3).

Secondly, in a platform setting, the platform provider has to address the information needs of the third-party providers that offer modules on the platform and the information needs of the platform and module consumers (C13).

Stakeholders:

The *API provider* has to identify the different roles in the *API consumer* organization that are involved in the API buying decision. In addition, the provider can collaborate with *sales & marketing* to create the website.

Implementation Hints:

Landing page. The landing page describes the purpose of the overall API portal and provides links to the product pages. Using the pattern `Role-based Marketing`, the landing page allows a visitor to choose between different views, e.g., a technical and a business view, or a platform consumer and a third-party developer view. Each view should carry a short description of the associated information.

Views. Each view contains information relevant to the respective stakeholder in a language that the stakeholder is accustomed to. The technical view focuses on the needs of developers, for example, by describing the user journey from registration to deployment, providing links to documentation early on, reducing marketing material to a minimum, and using illustrations to which developers are accustomed, e.g., UML diagrams (C3). Furthermore, the resources that the API portal provider compiles for developers should include technical specifications and code samples. The information can also include `Consumer-centric API Description`.

On the other hand, the business view addresses business users and presents more abstract information on potential use cases, lists advantages of using the platform, provides pricing information, and includes marketing material, e.g., `Customer Success Stories`. Additionally, the view can comprise an `Integration Partner Program` to point business stakeholders towards integration partners (C3; C13).

Similarly, in a platform setting, the view for a third-party developer should focus on technical documentation that allows the organizations to develop modules, sell them, and analyze their success.

Finally, the platform user view is mainly concerned with describing the platform's business value and additional services for the platform user (C13). Both platform views can include `Customer Success Stories`. The API provider can present the third-party developers with examples of successful modules developed by other third-party developers and the platform users with successful cases of platform usage (C13).

Content creation. The API provider can use the concept of *Personas*[1] (Cooper, 2004; Pruitt

[1]The concept of personas has emerged in the field of user-centered software design and denotes *"[...] descriptions of imaginary people constructed out of well-understood, highly specified data about real people"* (Pruitt & Adlin, 2005). The advantages of using Personas are that assumptions about users are made explicit, allowing the designer

& Adlin, 2005) to get a clear picture of the information requirements of different stakeholders. Furthermore, the API provider should collaborate with several internal stakeholders to ensure that the content created for the different external stakeholders is of high quality. For example, the API provider should involve Sales & Marketing in the design of marketing materials. All resulting resources have to be cross-checked to ensure consistency (C3).

Consequences:

Benefits:

- *Business information.* Information tailored to business stakeholders allows the business stakeholders to make better decisions. This is especially important in settings where integrating an API is of strategic relevance making it difficult for the API consumer to replace it later.
- *Technical information.* Separating the documentation into business and technical information makes it easier for developers to identify relevant information. The developers do not need to read any marketing information and can directly dive into technical documentation (C3).
- *Consumer types.* `Role-based Marketing` allows the API provider to address the information needs of different stakeholders in platform settings.

Drawbacks:

- *Business information.* The API consumers business stakeholder will potentially use the dedicated online documentation only for discovery. The actual buy decision usually happens only after personal contact and contract negotiations since the API consumer wants to create close ties and get service guarantees before choosing to use a strategically relevant API.
- *Effort.* It is laborious to create and maintain developer information and marketing resources, especially if they reference each other (C3). The API provider must ensure that the organization has enough resources to keep the documentation and marketing material up-to-date, especially if they release new API versions. Outdated documentation can lead to negative experiences for all stakeholders, business and technical.

Related Patterns within the Pattern Catalog:

The provider uses `Role-based Marketing` to organize information on the developer portal. Both business and technical stakeholders are interested in `Consumer-centric API Description` to understand which use cases or user stories an API can realize. Furthermore, information relevant for business stakeholders are `Customer Success Stories` and the

to focus on a specific type of users, and creating empathy and a shared understanding for the user (Pruitt & Adlin, 2005). Thus, the designer can address the needs of a user in a more targeted manner.

list of integration partners resulting from the `Integration Partner Program`. On the other hand, `Integration Guides` focus on the information needs of technical stakeholders.

Known Uses:

We observed the pattern in four cases:

First, the API initiative captured in C3 offers APIs to partner and third-party developers. The developer portal provides and separates technical and business information according to `Role-based Marketing`.

In C9, the organization offers a public marketplace for IoT applications. The organization provides core platform software, and third-party developers can build additional modules for the platform using APIs. Users can buy the core platform software and add pre-integrated modules according to their needs. The API marketplace offers dedicated documentation for two roles; users of the platform and its modules and the third-party developers that create these modules.

The API provider of C11 offers a financial service for API consumers to integrate into their systems. The API provider separates information for business stakeholders and developers. The information for business stakeholders comprises, e.g., use case descriptions, `Customer Success Stories`, and direct contact with sales. In comparison, the information for developers enables developers to request API credentials and links to API and SDK documentation.

Finally, in C13, the setting is the same as in C9, except that the platform is specific to the financial industry. Again, the marketing material is aimed at two groups; the platform software users and the third-party developers. The landing page of the API initiative shows advantages of being a provider or consumer on the marketplace.

Cross-case observations:

All these cases have in common that the API initiatives offer APIs to partners and the public. Also, the initiatives are in production and have 1-100 consumers. The dominant monetization strategy are the use of individual contractual agreements.

These characteristics are in unison with the observation that the APIs offer industry-specific functionality with a strategic impact on a consumer‘s business as opposed to commodity functionality. Strategic functionality is difficult to replace once the API consumer integrates it into its landscape and business plan. Thus, it makes sense that business stakeholders of the consumer organization are steering or at least involved in the decision of buying. Also, the API consumer relies on the API provider to deliver functionality with business impact, which makes close ties with the provider organization and service guarantees essential. Hence the parties create individual contractual agreements. The provision of strategic functionality for specific industries further limits the number of potential consumers, explaining why the API initiatives have between 1-100 consumers.

6.2. Customer Success Stories

Pattern Overview	
Name	`Customer Success Stories`
Alias	Inspire Case (C3); Customer Stories (C17)
Pattern Type	API Consumer-facing Pattern
Summary	A customer success story exemplifies an API consumer's successfully finalized use case or product implementation utilizing the provider's APIs (C3) with the aim to demonstrate an API's potential to future consumers.

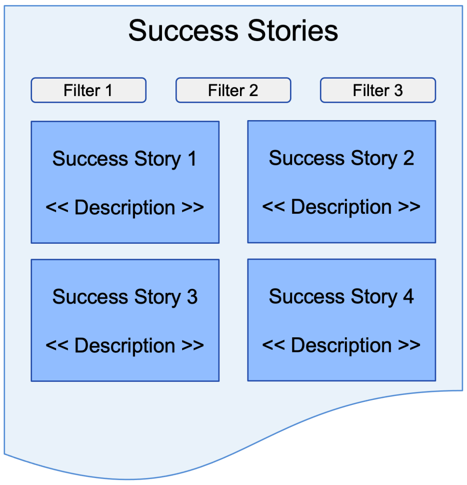

Figure 6.2.: An API provider presents `Customer Success Stories` on the API developer portal.

Context:

Especially in established organizations, technical and non-technical stakeholders are involved in buying an API (C3; C13). Thus, the API provider potentially already applies the pattern `Role-based Marketing` to address the information needs of different stakeholders separately. The API provider wants to create information relevant to business stakeholders, e.g., the upper management or procurement, to support these stakeholders with their buy decision (C3; C13).

Concern:

What information can the API provider make available for business stakeholders to support their understanding of an APIs value?

Forces:

- *API value.* Business stakeholders want to understand the use cases that an API enables and, thus, the value of the API before deciding to buy it (C3).
- *Trust.* API consumers want to gain an understanding of how reliable API providers and their APIs are before starting to use them.
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.

Solution:

An API provider can demonstrate an API's potential using success stories. A success story exemplifies an API consumer's successfully finalized use case or product implementation utilizing the provider's APIs (C3). Thus, a success story shows the potential of an API by describing a real-world example from a consumer perspective.

Stakeholders:

The *API provider* has to collaborate with the *API consumers* to identify and document successfully finalized use cases or product implementations. The provider can also involve *sales & marketing* in the creation and publication of `Customer Success Stories`.

Implementation Hints:

Approach. Success story creation comprises identifying successfully finalized projects. Next, the API provider has to collaborate with the API consumer to create and publish the story. *sales & marketing* can support the success story creation.

Contents. API providers can include different kinds of content in a success story. We observed the following contents (non-exhaustive):

- A description of the resulting use case or product, sometimes using visualizations or a video (C3; C9; C17).
- Tags categorizing the resulting use case or product (C17).
- A description of the consumer's business, usually with a link to their website (C3; C9; C17).
- A description of benefits for the partnering API consumer (C3), potentially using KPIs to substantiate these benefits (C9; C17).
- Quotes or whole interviews with the middle or upper management of the partnering API consumer emphasizing the advantages of the API (C3; C9; C17).
- A description of the final product's benefits for different end-users (C3).

- Links to the API(s) used to realize the success story (C3).
- Offers and options to replicate the resulting use case or product (C9).
- A contact form to contact the API provider (C9).

Success stories can be concise (C4), e.g., only mentioning the partnering API consumer and a quote, or quite extensive (C3; C9), containing almost all of the content types mentioned above.

API Portal. The API provider can display success stories differently on the API portal. For example, the API provider can show short success stories as banners on the landing page (C4). If the API provider decides to create more extensive success stories, it makes sense to create an overview page listing all success stories and linking to each success stories page (C3; C9). The overview page should include a filtering option if the API provider lists many success stories (C9).

Consequences:

Benefits:

- *API value.* Success stories enable business stakeholders to understand which use cases an API enables (C3). Thus, stakeholders get a better understanding of the value of an API for their business.
- *Trust.* Success stories show that an API provider has already successfully collaborated with other API consumers. Hence, success stories increase trust in the API provider organization and their ability to provide APIs reliably.

Drawbacks:

- *Effort.* It is laborious to create and maintain information and marketing resources, especially if the API provider has to coordinate success stories with existing API consumers.

Related Patterns within this Pattern Catalog:

The provider publishes `Customer Success Stories` on the developer portal, thus enabling API discovery. `Customer Success Stories` target business stakeholders according to `Role-based Marketing`.

Known Uses:

We observed the pattern in six cases.

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops. The API provider applies the pattern `Role-based Marketing` and uses `Customer Success Stories` as part of the information provided to business stakeholders. A success story comprises, among others, a description of the solution, a description of the partnering API consumer, the benefits of using the API for the consumer, a quote of a high-ranked role at the API consumer organization stressing the benefits of the APIs, and links to the integrated APIs.

Cases C4 represents the public API initiative of a financial software service provider. End users with a license for the API provider's software product automatically have access to the public API to enable customized integration with their systems. The API provider publishes banners with statements of consumers' executive employees about the successful integration of their products on their developer portal landing page.

In C9, the organization offers a public marketplace for IoT applications. The organization provides core platform software, and third-party developers can build additional modules for the platform using APIs. The platform provider publishes a gallery of success stories. The success stories range from concise one-pager descriptions of the use case and the benefits of using the platform to extensive blogs accompanying the integration project, including marketing material like videos. Due to the high number of success stories, the API provider offers filter facilities.

The API provider of C11 offers a financial service for API consumers to integrate into their systems. The API provider uses `Role-based Marketing` to differentiate between business stakeholders and developers. As part of the information for business stakeholders, the API provider presents pictures and consumer employee quotes describing the positive experience of using the API.

Stripe (C16) provides APIs for online payment processing[2]. Stripe presents a filterable gallery of success stories[3]. Most success stories follow a pre-defined structure comprising a description of the consumer's challenge, the solution, the results, and a quote from a consumer's employee[4]. Moreover, Stripe presents KPIs substantiating the consumer's benefits of using a product[5].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[6]. Twilio publishes a gallery of success stories with filtering facilities[7]. Filtering options comprise filtering for use cases, industry, region, and products[8]. A success story typically consists of a use case description, a description of the benefits for the consumer,

[2]https://stripe.com/en-de/about (accessed 20.12.2023).
[3]https://stripe.com/en-de/customers/all (accessed 20.12.2023).
[4]For example https://stripe.com/en-de/customers/agua-bendita (accessed 20.12.2023).
[5]For example https://stripe.com/en-de/customers/agua-bendita (accessed 20.12.2023).
[6]https://www.twilio.com/ (accessed 20.12.2023).
[7]https://customers.twilio.com/ (accessed 20.12.2023).
[8]https://customers.twilio.com/ (accessed 20.12.2023).

and quotes from the consumer employees[9].

Cross-case observations:

All the use cases have in common that their APIs are already in production, which makes sense since otherwise, it would not be possible to showcase past implementations. Also, while the API initiatives can focus on partner or public API consumers, the number of existing consumers is always higher than 20. Moreover, the API initiatives are API portals or marketplaces.

[9]For example https://customers.twilio.com/3321/birdeye/ (accessed 20.12.2023).

6.3. Newsletter

Pattern Overview	
Name	`Newsletter`
Pattern Type	API Consumer-facing Pattern
Summary	The API provider publishes summaries of changes to existing APIs (De, 2017) and other announcements related to APIs in a newsletter to keep current and potential future API consumers up-to-date.

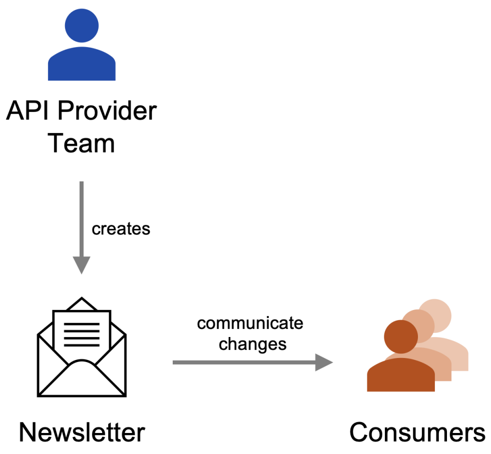

Figure 6.3.: The API provider creates a `Newsletter` to keep consumers up-to-date on changes.

Context:

API initiatives evolve, and API providers change APIs or publish new APIs. The API provider must inform (potential) API consumers about changes impacting existing integrations, enabling new integrations, or new business models (C4; C5).

Concern:

How can the API provider inform existing and potential API consumers about new APIs and changes to existing APIs?

Forces:

- *Breaking Changes.* Changes to existing APIs can impact API consumers. API consumers will abandon an API if their clients break due to unannounced changes too often.

- *New Business Models.* New APIs and changes to existing APIs might enable (potential) API consumers to realize new use cases or business models (C4; C5).
- *API Consumer Roles.* API consumer organizations consist of business and technical stakeholders. Communication with consumers should address the needs of each of these roles (C3).
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.
- *Privacy Concerns.* The collection of consumer data and the use of this data for marketing purposes is subject to strict privacy regulations.

Solution:

The API provider announces changes to existing APIs (De, 2017) and new APIs in a newsletter. Newsletter management comprises the collection and management of potential future and current API consumers' contact information, newsletter creation, and newsletter distribution.

Stakeholders:

The *API provider* can potentially collaborate with *sales & marketing* to design the newsletter. Furthermore, the provider should consult *legal* to ensure compliance with privacy regulations when collecting consumer information.

Implementation Hints:

Contents. A newsletter communicates information on changes to existing APIs (De, 2017; Medjaoui et al., 2018) and newly designed APIs. For example, newsletter content includes communication of changes to endpoints, use cases, or API products (C4; C5). Also, the API provider should communicate new features useful to the consumer (Medjaoui et al., 2018). However, not all changes are relevant for the API consumer, and communication should be limited to those changes that can affect the API consumers' client applications (Medjaoui et al., 2018).

Close collaboration with *sales & marketing* allows the API provider to use marketing best practices and tools for newsletter management, including contact management, newsletter creation, and distribution.

Contact data. The API provider collects and manages contact data of current and potential future API consumers in different ways. First of all, the developer portal should collect the contact information of all registered or subscribed developers (De, 2017; C4; C5). Furthermore, the API provider can collect contact information through support tickets. Also, the API provider should collect the contact information of anyone who contacts the API provider and shows interest in becoming an API consumer, even if no business relationship emerges at that point in time (C4; C5).

Consequences:

Benefits:

- *Breaking Changes.* API consumers can react to changes that affect their client applications on time if an API provider informs them ahead of time via a newsletter (Medjaoui et al., 2018).
- *New Business Models.* (Potential future) API consumers can stay up-to-date on new APIs and API changes that enable new use cases (Medjaoui et al., 2018) or business models. For example, a potential API consumer might did not adopt an API in the past because a certain functionality was missing (C4; C5). However, the newsletter informs this consumer that the functionality is now available, and they start using the API (C4; C5). Hence, using newsletters to announce changes can lead to increased adoption of APIs (C4; C5).
- *API Consumer Roles.* The API provider can customize newsletter content for business and non-business users.

Drawbacks:

- *Breaking Changes.* API consumers might miss the announcement of breaking changes in newsletters. Therefore, the API provider should use several communication channels to inform consumers about upcoming changes. Also, the API provider must communicate these changes ahead of time to allow the API consumers to react to them.

  Moreover, the API consumer should adopt a balanced API evolution approach that does not lead to breaking changes too often. Otherwise, the effort for API consumers to adopt clients might become too high, and they abandon an API, even if the API provider announces all changes in time.
- *Effort.* All communication with API consumers requires additional effort, especially if the API provider involves other departments like *sales & marketing* or *legal*.
- *Privacy Concerns.* API providers must ensure that the collection of API consumer data complies with current privacy regulations. The achievement of compliance creates effort for the API provider.

Related Patterns within this Pattern Catalog:

The API provider uses the `Newsletter` to inform current and potential consumers about changes to interfaces and contents of the developer portal.

Known Uses:

We observed the pattern in six cases.

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops. API consumers can subscribe to a newsletter on the API portal. The newsletter focuses on communicating new API products, but also API related events and success stories.

Cases C4 and C5 represent the public and partner API initiatives of a financial software service provider. End users with a license for the API providers software product automatically also have access to the public API to enable customized integration with their systems. The partner API allows other software providers to create integrations with their software products, thus managing more integrated solutions for end-users.
In these two cases, the API provider uses in-product and mailing list newsletters to promote changes in the API initiatives. In addition, contact data of (potential) API consumers is collected during business relation meetings, via support requests, and via the `Idea Backlog`. The API provider team has a dedicated marketing role that curates the content and manages the newsletters.

Next, in C13, the organization offers a public marketplace specific to the financial industry. Consumers can subscribe to a newsletter on the API portal's landing page. The newsletter communicates updates about new product features and API operations-related topics.

Stripe (C16) provides APIs for online payment processing[10]. API consumers can subscribe to a *Stripe Dev Digest*[11] to receive updates on APIs, products, SDKs, and client libraries. In addition, the newsletter communicates code examples, tutorials, and articles for developers to subscribers[12].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[13]. Twilio connects different kinds of support material for developers on their developer community website *Ahoy*[14]. On the Ahoy website, developers can subscribe to the *Ahoy Newsletter*[15] to receive an email newsletter on updates and other Twilio API-related content.

Cross-case observations:

All six cases have in common that the API initiatives are quite mature, meaning that the APIs are already in production and the number of API consumers is more than 20.

[10]https://stripe.com/en-de/about (accessed 20.12.2023).
[11]https://go.stripe.global/dev-digest (accessed 20.12.2023).
[12]https://go.stripe.global/dev-digest (accessed 20.12.2023).
[13]https://www.twilio.com/ (accessed 20.12.2023).
[14]https://www.twilio.com/ahoy (accessed 20.12.2023).
[15]https://www.twilio.com/ahoy (accessed 20.12.2023).

Furthermore, all API initiatives are for partners or the public. Thus, communication of changes should follow a structured approach that reaches all API consumers.

6.4. Consumer-centric API Description

Pattern Overview	
Name	`Consumer-centric API Descriptions`
Pattern Type	API Consumer-facing Pattern
Summary	The API provider describes the API products functionality from a consumer perspective as use cases or user stories addressing a consumer's business need.

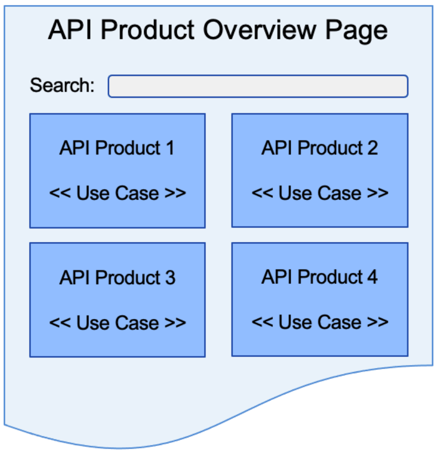

Figure 6.4.: A `Consumer-centric API Description` lists the major use cases or user stories of the APIs on the developer portal.

Context:

The value of an API depends on API consumers using it (Medjaoui et al., 2018). API consumers look for APIs to solve specific business needs. Therefore, an API provider has already applied the pattern `API-as-a-Product` and created API products. However, API consumers need to find and understand the API's functionality before they can start using it (De, 2017; Medjaoui et al., 2018). Thus, the API provider has to ensure that the API consumer can discover and understand the business value of the API product.

Concern:

How can the API provider describe API products to ensure discoverability and comprehensibility for API consumers?

Forces:

- *Different roles:* Within an API consumer organization, different roles are involved in the search and decision process for new applications and APIs. Such roles comprise technical and business roles who might not have a good understanding of APIs (De, 2017; C13). Hence, a solution should enable each of these roles to easily understand the value of an API or API product's functionality.
- *Searchability.* Especially if the API provider operates a marketplace, the API consumer has to search a long list of available APIs. In such settings, the API provider should support the API consumer with the search for suitable APIs.
- *Marketing information.* Developers do not like to read marketing information (Jacobson et al., 2012). Instead, API documentation should focus on facts. Hence, a solution should not add unnecessary marketing information to the API documentation.
- *Provider Effort.* The API provider has limited resources. Therefore, a solution should create little additional effort for the API provider.

Solution:

The API provider describes the API products functionality from a consumer perspective as use cases or user stories addressing a consumer's business need. As part of the API portal, an API product overview page lists all API products and their use cases or user stories.

Stakeholders:

The *API provider* can collaborate with *sales & marketing* to publish `Consumer-centric API Description` on the developer portal.

Implementation Hints:

Use cases and user stories. After bundling APIs into products according to `API-as-a-Product`, API providers describe the functionality of the products as use cases[16] and user stories[17] from a consumer perspective. Therefore, the API provider specifies the overall purpose of an API

[16]A use case describes the functionality of a system that is of value for users and other stakeholders (OMG UML, 2017; Spichale, 2017). Use cases enable the definition of requirements for a system with the API consumers' goals in mind, but also the description of an existing systems functionality (OMG UML, 2017). In addition, use cases focus on the behavior of a system and are often described as interactions between the system and its users (OMG UML, 2017). Therefore, Cockburn (2000) defines a use case as *"[...] a contract between the stakeholders of a system about its behavior"* (Cockburn, 2000, p. 2).

[17]In the context of agile development, *"A user story describes functionality that will be valuable to either a user or purchaser of a system or software."* (Cohn, 2004). However, agile teams use user stories in the context of requirements definition and consisting of a written description, discussions about the description, and acceptance tests (Cohn, 2004). We assume that discussions and acceptance tests happened before publishing the API product, and the user stories describing an API product or use case reflect that process.

and the core functionality of a product that addresses consumers' business needs (Spichale, 2017). These descriptions of the functionality do not comprise any technical details (OMG UML, 2017). However, the high-level description links to more detailed information in the documentation (Spichale, 2017).

API portal. The API provider creates an overview of all products on a product overview page in the developer portal (De, 2017). The product overview page can list products in different formats, but in most cases, API providers represent products as tiles, including the product's name and the user story or stories it implements. In addition, if the API portal lists many products, the API provider can implement search or filter options, e.g., a filter for geographic regions in which an API product is available.

Consequences:

Benefits:

- *Different roles:* The description of an API or API product's functionality as use cases or user stories does not entail technical details (OMG UML, 2017) but describes the functionality from a consumer perspective. Hence, different roles, including business roles, can understand the value of the functionality for an organization.
- *Searchability.* API consumers usually search for an API that fulfills a business need. Consumer-centric API descriptions enable API providers to offer search and filter options that allow API consumers to search more efficiently for APIs that offer a certain functionality (De, 2017).
- *Marketing information.* Consumer-centric API descriptions do not entail any technical information. However, the developers need to understand the functionality of the API or API product, which the API provider can convey through use cases or user stories. Nevertheless, the consumer-centric API descriptions should be as short as possible.

Drawbacks:

- *Provider Effort.* The identification of suitable use cases and user stories that help an API consumer to understand an API's functionality creates effort for the API provider (Medjaoui et al., 2018). However, the API provider might already create these consumer-centric descriptions during the design and development of the API or API product (Spichale, 2017). The API provider can easily reuse the use cases or user stories in such cases.

Related Patterns within this Pattern Catalog:

The provider publishes `Consumer-centric API Description` on the developer portal, thus enabling API discovery. `Consumer-centric API Description` target business as well as technical stakeholders according to `Role-based Marketing`.

In addition, `Consumer-centric API Description` defines user stories or use cases which is a prerequisite for `Integration Guides` since they use user stories or use cases to guide integration documentation.

Other Related Patterns:

In Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Lübke et al. (2019), the pattern `API Description` defines what knowledge should be shared between the API provider and consumer. The solution states that the API provider should include different kinds of knowledge, including technical-, quality-, and organization-related information, with API consumers. Furthermore, Lübke et al. (2019) presents a template for API documentation. The template includes the definition of "user stories and quality attributes (design forces)." Hence, the pattern `Consumer-centric API Description` could be part of the pattern `API Description`.

Similarly, the pattern `Consumer-centric API Description` could support the realization of the `Interface Description` pattern presented by Völter et al. (2004).

Known Uses:

We observed the pattern in four cases:

The API portal provider of the automotive organization in C3 makes vehicle data accessible to business and private consumers. The API provider describes each API product with a use case on their developer portal.

The API provider of C11 offers a financial service client library for API consumers to integrate into their websites and online shops. The API provider uses use cases to describe the API's functionality and structure the documentation.

Stripe (C16) provides APIs for online payment processing[18]. Stripe uses use cases to show the value of the functionality of their API. For example, Stripe describes the "Payments" API with the high-level use case "Build a web or mobile integration to accept payments online or in person"[19]. Moreover, Stripe breaks down the overall use case into more specific use cases, e.g., "Accept online payments: Build a payment form or use a prebuilt checkout page to accept online payments"[20]. These use cases structure the documentation of the payment APIs from a user perspective and link to integration guides that support API consumers with the use case implementation.

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[21]. Twilio describes subsets of the channel APIs with high-level use

[18]https://stripe.com/en-de/about (accessed 20.12.2023).
[19]https://stripe.com/docs (accessed 20.12.2023).
[20]https://stripe.com/docs/payments (accessed 20.12.2023).
[21]https://www.twilio.com/ (accessed 20.12.2023)

cases[22]. For example, "Programmable Messaging" covers the high-level user story *"Send and receive messages on SMS, MMS, and WhatsApp"*[23], which API consumers can realize with the "Programmable Messaging API"[24], the "WhatsApp Business API"[25], and the "Conversations API"[26].

Cross-case observations:

The cross-case observations are similar to those made for the pattern `API-as-a-Product`. API initiatives within our case base that describe API products as user stories are public initiatives. Describing API products as user stories for public initiatives makes sense since a major goal of user stories is to make the APIs easily discoverable for a broad range of consumers with whom no prior relationship exists. Also, the API initiatives are quite mature, with several APIs in production and several existing consumers. Additionally, three out of four API initiatives belong to software or IT service provider organizations with experience designing and managing intangible products. Finally, almost all of the API products are monetized and not for free.

[22]https://www.twilio.com/products (accessed 20.12.2023).
[23]https://www.twilio.com/products (accessed 20.12.2023).
[24]https://www.twilio.com/messaging (accessed 20.12.2023).
[25]https://www.twilio.com/whatsapp (accessed 20.12.2023).
[26]https://www.twilio.com/messaging (accessed 20.12.2023).

6.5. Integration Guide

Pattern Overview	
Name	`Integration Guide`
Alias	Tutorials (Medjaoui et al., 2018; Spichale, 2017; C17), Usage Guides (C17), Scenario documentation (Medjaoui et al., 2018), Walkthroughs (De, 2017), Cookbook (Spichale, 2017)
Pattern Type	API Consumer-facing Pattern
Summary	An integration guide documents the implementation of common functionality using step-by-step instructions (Spichale, 2017) to reduce consumers' effort implementing the specific functionality (Medjaoui et al., 2018).

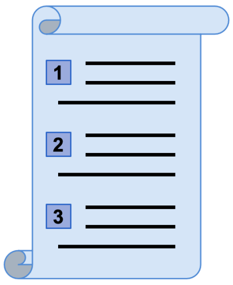

Figure 6.5.: The `Integration Guide` provides step-by-step instructions to implement a specific functionality.

Context:

An API provider uses APIs as the primary channel to make a product accessible to external consumers or the API is a major feature of a software product. Moreover, the API provider has published technical reference documentation. The number of API consumers is high, and the API is in a relatively mature state (Medjaoui et al., 2018). Many consumers want to use the API to realize the same functionalities. However, realizing these functionalities is challenging and creates effort for the API consumers. Especially in markets with several APIs and solutions for the same functionality, API consumers consider the effort necessary to implement functionality when choosing an API (Lübke et al., 2019).

Concern:

How can the API provider support API consumers with the implementation of common API functionalities?

Forces:

- *Knowledge Transfer.* API consumers cannot use an API without the API provider transferring the necessary knowledge to them (De, 2017; Jacobson et al., 2012; Lübke et al., 2019; Medjaoui et al., 2018; Spichale, 2017). If API providers only publish reference documentation, API consumers might make assumptions (Lübke et al., 2019), try to reverse engineer the APIs behavior (Lübke et al., 2019), implement bugs (Medjaoui et al., 2018), or fail to implement a functionality altogether. Thus, ambiguities can lead to violations of the information hiding principle and increase the API consumers' effort to maintain a client application (Lübke et al., 2019). Therefore, a solution has to communicate relevant knowledge to the API consumers.

- *Consumer Effort.* API consumers do not have the time nor the willingness to work through a lot of documentation (Jacobson et al., 2012). Thus, documentation is a *"key success factor for an API"* (Jacobson et al., 2012, p.100) and can be a competitive advantage compared to other APIs with the same functionality (Jacobson et al., 2012; Medjaoui et al., 2018).

- *Developer Experience.* Documentation is part of the developer experience (Medjaoui et al., 2018; C11). Moreover, the usability of documentation influences consumers' decision to use an API (Lübke et al., 2019) and therefore affects the APIs strategic value (Medjaoui et al., 2018), and social adoption (De, 2017). Fewer API consumers will use an API that is difficult to learn or use due to insufficient documentation (Medjaoui et al., 2018).

- *Support.* The quality of documentation can influence the number of support requests (Spichale, 2017). The more knowledge the documentation transfers to the API consumer, the less support the API consumers need.

- *Beginners and Advanced User.* Consumers starting to use an API have different information needs compared to advanced consumers (Spichale, 2017). Nevertheless, a solution should cater to the information needs of all consumer groups (Spichale, 2017).

- *Consistency.* API consumers expect consistency and accuracy across all types of documentation (Lübke et al., 2019). Errors or inconsistencies in the documentation can drive API consumers to abandon an API (Medjaoui et al., 2018).

- *Provider effort.* API providers have limited resources to design and maintain API documentation. Thus, a solution should require little additional effort or investment for API providers.

Solution:

The API provider creates and publishes integration guides that support API consumers with the implementation of common functionality using step-by-step instructions (Spichale, 2017). Such an integration guide usually comprises a description of the functionality, a specification of activities necessary before implementing the functionality, the steps for implementing the functionality enriched with copyable code snippets, and a guide to testing the implemented functionality. Hence, an integration guide reduces consumers' effort to implement the specific functionality (Medjaoui et al., 2018).

Stakeholders:

This pattern addresses a concern of an *API provider*. The provider has to communicate with *API consumers* to decide for which functionality the effort of creating integration guides is justified. Furthermore, the provider should communicate with consumers to identify improvement potentials for existing integration guides. In addition, the provider can involve *sales & marketing* to enhance the design of integration guides on the developer portal.

Implementation Hints:

Structure. An integration guide provides step-by-step directions that allow an API consumer to implement the solution to a specific consumer problem (Medjaoui et al., 2018; Spichale, 2017). An integration guide first specifies the functionality, e.g., using a user story or use case description[27], that the integration guide helps to implement (C11; C16; C17). These specifications should be easy to understand and can include additional resources, e.g., visualizations (C11). Also, the integration guide introduces necessary domain knowledge in a straightforward and easy-to-understand manner (C11).

Afterward, the integration guide lists tasks or requirements that API consumers must fulfill before starting to code (C11; C16; C17). Such preceding tasks include, for example, the registration of a user account (C17), the retrieval of relevant authentication information (Spichale, 2017; C11; C17), or the download of specific libraries (Spichale, 2017; C11; C16).

Next, the integration guide describes the steps to implement the functionality (C11; C16; C17). If applicable, the descriptions are enriched with code snippets that the API consumer can copy (De, 2017; Spichale, 2017; C11; C16; C17). Such code examples should take into account the programming languages that most API consumers use (Spichale, 2017). The code snippets implement the logic necessary to realize the functionality and are optimized

[27]In the context of agile development, *"A user story describes functionality that will be valuable to either a user or purchaser of a system or software."* (Cohn, 2004, p. 4). Agile teams use user stories for the definition of requirements. A user story consists of a written description, discussions about the description, and acceptance tests (Cohn, 2004).

A use case describes the functionality of a system that is of value for users and other stakeholders (OMG UML, 2017; Spichale, 2017). Use cases enable the definition of requirements for a system, but also the description of an existing systems functionality (OMG UML, 2017). Moreover, use cases focus on the behavior of a system and are often described as interactions between the system and its users (OMG UML, 2017). Therefore, Cockburn (2000) defines a use case as *"[...] a contract between the stakeholders of a system about its behavior"* (Cockburn, 2000, p. 2).

Thus, in summary, user stories and use cases describe the functionality of a system. Also, both do not disclose any technical or implementation details (OMG UML, 2017). However, we understand use cases as descriptions of functionality derived from users' goals with a focus on the system and its behavior. In comparison, user stories focus on the consumer perspective by describing the functionality as a solution to a consumer's problem from the consumer's point of view.

for quality properties like performance (C11). The `Integration Guide` also explains how API consumers must adapt the code snippets, e.g., where to replace placeholders with authentication information (C11; C16; C17).

An alternative approach is to provide querying tools embedded into the documentation (Jacobson et al., 2012). Such tools allow API consumers to test different calls and thus gain an understanding of the API without the need to go through setup activities (Jacobson et al., 2012).

After the implementation, the integration guide supports testing implemented functionality (C11; C16).

Throughout all parts of the integration guide, the API provider includes links to other resources that provide more detailed information not necessarily relevant for implementing the functionality (C11; C16; C17). Moreover, the API provider can add best practices or tips that support the API consumer, e.g., the best placement of buttons on a website (C11).

In summary, an integration guide captures the knowledge, steps, and code snippets necessary to realize a certain functionality from beginning to end in an easy-to-understand manner. Hence, the guide relieves API consumers from the effort of repetitively developing the same or similar workflows. The provider can offer basic "Hello World" guides and more advanced tutorials building on the basic guides (Spichale, 2017).

<u>APIs and Libraries.</u> The API provider can offer integration guides for the Web API itself, client libraries, or both (Medjaoui et al., 2018). In many cases, the API consumer can choose the preferred programming language at the beginning of the integration guide (C16; C17).

<u>Publication Platform.</u> The API provider should make integration guides available on the developer portal since it should be the single source for all information related to an API (De, 2017; Jacobson et al., 2012; Lübke et al., 2019; Medjaoui et al., 2018; C11; C16; C17).

<u>API Changes.</u> With every change of an API, the API provider has to validate or adapt the associated integration guides (De, 2017; Medjaoui et al., 2018). Therefore, an API provider should design processes and execute tests to ensure that integration guides are still up to date after an API change (De, 2017; Lübke et al., 2019). In addition, if the API provider offers several versions of an API, integration guides need to indicate which version they support clearly.

<u>Continuous improvement.</u> Tutorials are part of the onboarding experience for API consumers. Therefore, the API provider should automatically and manually gather data and feedback on the API consumers' use of tutorials to continuously derive and implement improvements (Medjaoui et al., 2018).

<u>Documentation requirements.</u> Also, an API provider can centrally prescribe documentation requirements that each API team needs to fulfill (Medjaoui et al., 2018). For example, a potential requirement is that the documentation includes an integration guide.

Consequences:

Benefits:

- *Knowledge Transfer.* It depends on the complexity of the API or functionality that an API consumer wants to implement as well as on the experience of the API consumer if integration guides add value for API consumers (Spichale, 2017). For example, in settings where the API and functionality are complex, or most API consumers have little experience with the API, reference documentation might not be sufficient to enable the API consumers to implement the functionality. Integration guides describing each step necessary to implement specific functionality are useful in such settings. Hence, integration guides support knowledge transfer and remove ambiguities.
- *Consumer Effort.* Insufficient documentation leads to increased learning, experimentation, development, and testing effort for API consumers (Lübke et al., 2019). An integration guide can decrease the time and effort for implementing a specific functionality.
- *Developer experience.* Human-readable documentation delivers the most popular learning experience (Medjaoui et al., 2018). Furthermore, step-by-step guides help API consumers to learn a new API quickly. Thus, integration guides can improve the developer experience.
- *Support.* Integration guides support API consumers with the solution of recurring issues and the implementation of standard functionality (Spichale, 2017). Thus, the API consumers need less support from the support team.
- *Beginners and Advanced User.* A beginner's learning curve should be as flat as possible (Spichale, 2017). In addition, app developers look for quick-start guides and tutorials when they start using an API (De, 2017). Thus, integration guides are handy for new API consumers just learning the API (Medjaoui et al., 2018; Spichale, 2017).

Drawbacks:

- *Knowledge Transfer.* In settings where the API and functionality are simple, or most API consumers have much experience with the API, reference documentation can be sufficient to remove ambiguities and enable API consumers to implement the functionality.
- *Beginners and Advanced User.* Advanced users are probably less interested in integration guides for common use cases but instead in best practices and reasons for specific design decisions (Spichale, 2017). However, the integration guides can link to such documentation.
- *Consistency.* If different types of documentation provide redundant information, the effort of creating and maintaining the documentation increases (Lübke et al., 2019). In addition, since the reference documentation and integration guides comprise partially redundant information, integration guides can increase the likelihood of inconsistencies between documentation types.

- *Provider effort.* The creation and maintenance of integration guides are effortful and costly (Lübke et al., 2019; Medjaoui et al., 2018). The API provider organization experts must manually generate and review the integration guides (Jacobson et al., 2012) and test associated code snippets. Furthermore, the API provider needs to check and potentially update the integration guides with every API change (De, 2017; Medjaoui et al., 2018). To prevent errors or inconsistencies, the API provider should design processes and tests that ensure that the documentation is always up to date (De, 2017; Lübke et al., 2019). However, the API provider needs to invest effort into setting up such processes. In addition, the API provider should continuously try to improve the integration guides. In comparison, reference documentation is easier to design and update, especially if the API provider uses tools to generate the documentation automatically (Lübke et al., 2019). Thus, the API provider has to carefully evaluate if the number of API consumers benefiting from the integration guide justifies the additional effort (Medjaoui et al., 2018).

Related Patterns within this Pattern Catalog:

`Integration Guides` are published on the developer portal and target technical stakeholders according to `Role-based Marketing`.

Moreover, `Integration Guides` provide step-by-step instructions on how to integrate user stories or use cases as defined in `Consumer-centric API Description`.

Other Related Patterns:

In Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Lübke et al. (2019), the authors introduce the pattern `API Description`. The pattern tackles what knowledge an API provider should share with API consumers and how to document the shared knowledge. The solution states that the API provider should include different kinds of knowledge, including technical-, quality-, and organization-related information, with API consumers. In addition, it is crucial to share basic facts about the API, like network addresses and call structures, and provide additional information on, e.g., invocation sequences. Such information reduces the API consumers' effort to understand and use the API. However, how much knowledge a description comprises should depend on a specific API's market dynamics and development culture. Hence, an `Integration Guide` could be part of the pattern `API Description`.

Similarly, the pattern `Consumer-centric API Description` could support the realization of the `Interface Description` pattern presented by Völter et al. (2004).

Known Uses:

We observed the pattern in five cases:

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops. The API provider presents a step-by-step guide on implementing the most common functionality for each API. These guides comprise code examples.

Case C4 represents a software service provider's public API initiative. End users with a license for the API provider's software product automatically have access to the public API to enable customized system integration. The API provider publishes an integration guide on the developer portal. The integration guide describes the most common calls to each API endpoint, including code examples.

The API provider of C11 offers a financial service client library for API consumers to integrate into their systems. This API provider offers integration guides that support API consumers with the realization of specific functionality. The integration guides consist of step-by-step directions informing the API consumer about relevant domain knowledge, code snippets, explanations on how to adapt the code, and guidance on testing.

Stripe (C16) provides APIs for online payment processing[28]. To support API consumers with API integration, Stripe offers integration guides. For example, if API consumers want to "accept a payment"[29], they can simply follow the listed steps. Each of these steps has a description potentially enriched with code snippets. The steps also comprise setup and testing guidance. Moreover, since Stripe provides several `Client Library`, the API consumer can choose the code snippets in different programming languages. Thus, Stripe guides the API consumer through the integration of the "accept payment" functionality with a detailed description.

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[30]. In addition to other services, Twilio offers several channel APIs, each focused on a specific communication channel[31]. Focusing on the "Programmable Messaging API" for sending and receiving SMS and MMS messages, the documentation includes a set of *"Tutorials"*[32] that describe the integration of the APIs functionality into a client application.

Cross-case observations:

The five cases have a lot in common. First, they represent mature API initiatives with a high number of API consumers. Thus, investing additional effort into documentation makes sense since good documentation can reduce the need for personal interactions and thus enables the scaling of API initiatives. Also, all API initiatives have predefined monetization schemes and high levels of self-service automation. Finally, in these cases, the API is the primary channel to make a software product accessible, or it is a significant feature of the software product.

[28]https://stripe.com/en-de/about (accessed 20.12.2023).
[29]https://stripe.com/docs/payments/accept-a-payment?platform=web&ui=checkout#set-up-stripe (accessed 20.12.2023).
[30]https://www.twilio.com/ (accessed 20.12.2023).
[31]https://www.twilio.com/products (accessed 20.12.2023).
[32]https://www.twilio.com/docs/api (accessed 20.12.2023).

6.6. Onboarding Self-service

Pattern Overview	
Name	`Onboarding Self-Service`
Pattern Type	API Consumer-facing Pattern
Summary	An onboarding self-service automates (parts of) the API onboarding process by allowing API consumers to choose a monetization plan, register a user account, generate authentication credentials, and register a finalized client application without interacting with API provider team members.

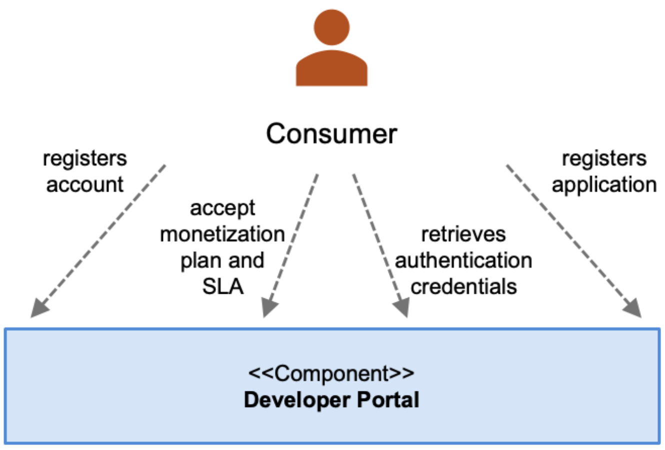

Figure 6.6.: If the API provider offers an *Onboarding Self-service*, the consumer does not have to interact with the provider before starting to use an API.

Context:

API consumers want to use an API as fast as possible after discovery. However, some interaction between consumers and the API provider is necessary before API consumers can start using an API. Such interactions comprise the negotiation of monetization schemes and service levels as well as the execution of identification and authentication mechanisms.

Concern:

How can API providers enable API consumers to start using an API as fast as possible?

Forces:

- *Fast access.* An API provider should implement security measures, e.g., API consumer identification, authentication, and authorization (Jacobson et al., 2012). These security mechanisms require an interaction between the API provider and the API consumer. However, an API consumer does not want to wait for manual responses from the API provider (Jacobson et al., 2012). If the onboarding processes take too long or are too complicated, API consumers might lose interest in an API (Jacobson et al., 2012). Ideally, the registration process should take less than five minutes (Jacobson et al., 2012). Hence, user registration, application authentication, and application review processes impact the developer experience (Jacobson et al., 2012). Especially in the context of APIs that follow the `API-as-a-Product` approach, the API provider should put effort into creating a fast onboarding experience for API consumers (Medjaoui et al., 2018).

- *Transparency.* An API consumer will only enter a business relationship with an API provider if the monetization scheme and the SLAs are transparent and attractive (C13).
- *Negotiations.* Especially in partner API initiatives, API consumers often do not want to accept standard monetization schemes and SLAs. Instead, they want to negotiate special monetization and service levels for using an API (C13).
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.

Solution:

An API provider can automate some or all of the interactions between the API provider and API consumer necessary to allow the API consumer access to the API. First, instead of negotiating monetization schemes, the API provider can present predefined monetization plans to the API consumer. Similarly, the API provider can create predefined service levels and other contractual agreements. Also, the API provider can provide the means for the API consumer to register a user account and access credentials for authenticating API requests via the developer portal. Thus, the API consumer can start using the API within minutes without the need to spend time and effort to interact with the API provider.

Stakeholders:

Within the *API provider* team, the *API portal provider* has to integrate the self-service functionality in the developer portal. In addition, the *API portal provider* can collaborate with *legal* and *finance & controlling* to design predefined monetization plans, SLAs, and other contractual agreements.

Implementation Hints:

Approach. Self-service means that the API provider automates tasks necessary to start using an API that would usually require an interaction between the API provider and the API consumer. The API provider team designs the API consumers workflow from registration of an account to the use of an API (De, 2017; Medjaoui et al., 2018). The consumer workflow comprises, for example, account registration (De, 2017), acceptance of monetization schemes, SLAs, and other contractual terms, potentially a client application review, and the application authentication (Jacobson et al., 2012). The API consumers can trigger these tasks by entering information into the self-service system. Self-service features are usually part of the API developer portal (De, 2017; Jacobson et al., 2012).

In the following, we will present each activity that the API provider can automate as a self-service in more detail, i.e., registration, monetization, terms guiding the business relationship, client application review, and application authorization. The activities can be in a different order than the order in which they are described.

Registration. The API consumers can register with their personal or company information on the developer portal to create a user account (De, 2017). The API consumer must provide billing information if the API is not free (C9). Furthermore, especially in the finance sector, API consumers sometimes need to validate their real-world identity (C11; C16). However, the registration should only require the API consumer to provide as little information as possible not to drive the consumers away (De, 2017).

Potential authentication methods include username and password (e.g., HTTP basic authentication), OAuth, SAML, or X.509 certificates, two-way SSL (Secure Socket Layer), or WS-security specifications (Jacobson et al., 2012). In most cases, the API provider enables API consumers to register new accounts with an email address and a password (C3; C9; C11; C16).

Monetization. An API provider has to offer predefined monetization schemes and service levels to realize a self-service. Often API providers offer several plans (De, 2017) that differ regarding the accessible functionality, the number of allowed calls, or the different service levels. An API consumer can choose one of these predefined plans (C3; C9; C11; C16). The API provider should collaborate with the *finance & controlling* department to calculate business plans and create monetization schemes. However, if an API consumer does not want to accept a predefined plan, self-service is impossible.

Terms guiding the business relationship. Similarly, the API provider should create and publish a `Service-Level Agreement (SLA)`, terms of use, and any other agreements that determine the business relationship between the API provider and API consumer. These agreements should be acceptable for most API consumers, who have to accept them as they are. The API provider should collaborate with the *legal* department to design and review the agreements.

Client application review. In some cases, the API provider only hands out authentication credentials for a sandbox environment right away. The API consumer only gains access to the live API only after a client application review (C11; C14). The API consumers usually trigger an application review in the developer portal (De, 2017), e.g., by filling in a form describing the application. An API provider might decide to automatically or manually review and approve the consumer application (De, 2017). If the review does not raise any concerns or questions, the API consumer receives the authentication credentials for the production environment. However, if there are any questions or issues, the API provider and API consumer must communicate to resolve the problem or uncertainty (C3; C11).

Application authorization. An API key (De, 2017; Jacobson et al., 2012) allows the API to know which application and thus which developers use it (Jacobson et al., 2012). The API provider blocks requests to the API without valid authentication credentials (Jacobson et al., 2012; Spichale, 2017). The API provider can automate the generation and management of authentication credentials in the developer portal instead of communicating these credentials manually (C3; C4; C14; C16).

Full and partial self-service. In general, an API provider can decide to offer self-service for all or only some of the tasks described above. For example, for public API initiatives, we often observed that the API provider offers API consumers self-service capabilities covering monetization, terms guiding the business relation, registration, and authentication only for test environments. Thus, the API consumers can start using the test environment within minutes. Still, the API provider restricts access to the production environment without prior application review, which can take several days (C3; C11; C14).

However, we observed that sometimes API consumers do not want to accept standard monetization schemes and other contractual conditions but instead like to negotiate individual agreements, especially in partner API settings. Thus, it takes time until the API consumer can access the API. However, afterward, they also rely on self-service for registration and authentication (C13).

Continuous improvement. A measure to improve the onboarding experience is to monitor the API consumer behavior during onboarding, e.g., measuring the time it takes the consumer to complete a task or identifying steps where the consumer fails or stops using the API (Medjaoui et al., 2018).

Consequences:

Benefits:

- *Fast access.* A self-service allows API consumers to use an API in minutes without waiting for the API provider to perform manual activities (C2). The advantages of a good onboarding experience are reduced support costs and increased customer satisfaction (Jacobson et al., 2012; C14).
- *Transparency.* The API provider can formulate clear and attractive standard monetization schemes and SLAs for API consumers. Such standard monetization schemes and SLAs enable the API consumer to quickly and easily enter a business relationship with an API provider.

Drawbacks:

- *Negotiations.* The API provider cannot offer full self-service when API consumers do not accept standard monetization schemes and SLAs. However, in these cases, the

API provider can offer at least a partial self-service for authentication and application registration (C13).

- *Effort.* The API provider has to create attractive standard monetization schemes and agreements, which require effort and collaboration with different departments, e.g., *legal* and, *finance & controlling* (C13). Also, an API provider must implement a user interface for the API consumer to register and manage the account and authorization credentials (Jacobson et al., 2012). However, commercial API gateways usually offer these functionalities as out-of-the-box features (Jacobson et al., 2012; Medjaoui et al., 2018; Spichale, 2017).

Related Patterns within this Pattern Catalog:

After discovery of an API, the `Onboarding Self-service` allows consumers to quickly access the `Web API`, `Client Library`, or `Frontend Venture`.

Other Related Patterns:

In Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018), the authors present two patterns related to the pattern `Onboarding Self-service`. First, the pattern `API Key` is a unique, provider-allocated key that the API consumer includes in each request to the API. The `API Key` enables the identification and authentication of client applications. Thus, API providers use `API Keys` to ensure that only registered consumers can make successful API calls. Also, the `API Key` enables monitoring calls, allowing for pay-per-call monetization or enforcing different access rights for each consumer. An additional advantage of an `API Key` is that API consumers do not have to include account login credentials in each call to the API. Hence, the `API Key` decouples consumer accounts from consumer roles, allowing for more fine-grained permissions management. Also, the API consumer can easily replace an `API Key` in case of a security break. Alternative names for the pattern `API Key` are *Access Token* and *Provider-Allocated Client Identifier*.

Similarly, Richardson (2019) and Richardson (n.d.) present the pattern `Access Token`. An `Access Token` identifies a consumer sending requests to a service.

Hence, an `API Key` (Stocker et al., 2018; Zimmermann et al., n.d., 2022) or `Access Token` (Richardson, n.d., 2019) can support the realization of the pattern `Onboarding Self-service`.

Known Uses:

We observed the pattern in nine cases:

In the case of C2, the API provider offers simulation and modeling algorithms to analyze energy data. The API provider aims to provide a full onboarding self-service for API consumers. API consumers are requested to provide feedback if the onboarding self-service is not self-explanatory to enable the API provider to improve the consumer experience.

The API portal provider of an automotive organization (C3) makes vehicle data accessible to business and private consumers. The organization offers different options for accessing APIs. The first potential option is a sandbox with limited test data. The second option is private use of the API using personal car data. For both, the sandbox and the private use of an API, the developer has to register on the portal using an email address and a password. Afterward, the developer subscribes to relevant APIs, receives an API key or OAuth credentials, and can start using the APIs. The third option for accessing an API is a business option. In this case, the API consumer has to contact the API provider to negotiate a contract. As soon as the contractual agreement is in place, the API consumer can access all data provided by the API. However, before the application using the API goes live, the API provider reviews it. Therefore, the API consumer can only publish the application after the approval of the API provider. Hence, the organization offers full self-service in some situations and only partial self-service in others.

The API provider of C4 provides a financial services software product. In addition, the API provider offers an API that allows users of the software product to integrate the software product with their system landscape. First, API consumers must register on the developer portal to create a user account. Afterward, the API consumer generates an API key in the developer portal. To be successful, the API consumer must include the API key as an authentication credential in the API requests. With the API key, the API consumer can access all available APIs.

In C9, the organization offers a public marketplace for IoT applications. The organization provides core platform software, and third-party developers can build additional modules for the platform using APIs. These developers can register for a free account with an email address and a password but also need to provide further information about their organization to complete the registration process successfully. Moreover, the API consumer has to agree to a standard privacy notice and terms & conditions. The features of a free account are limited, but developers can run applications on the IoT platform. While API provider can set up a free account as a complete self-service, paid accounts require interaction between the provider and consumer.

The API provider of C11 offers a financial service for API consumers to integrate into their systems. First, the API consumer has to register for a developer, a personal, or a business user account. After registration, a developer dashboard issues a client ID and credentials that identify and authorize the consumer's application to access the sandbox. A switch from the sandbox to the production environment requires the API consumer to request credentials for the live API. Hence, an API consumer can access the sandbox environment within minutes since the API provider provides self-service capabilities for identification and authorization.

In C13, the organization offers a public marketplace for insurance applications. The API marketplace provider allows third-party API providers to integrate additional modules via APIs and publish these modules on the marketplace. These modules can be accessible via API or as apps. Users of the platform can buy the modules according to their needs. Thus, the marketplace provider has to enable third-party API providers and API consumers to access the marketplace. First, no matter if a user becomes a third-party API provider or an API consumer, they have to register with some basic personal information, an email, and a password. As part of the registration, the user must confirm the email address and accept the marketplace's terms and conditions with a click on the respective checkbox. Next, the user has to choose between a third-party module provider and a consumer account. The documentation describes the sign-up steps for each of these roles following a `Role-based Marketing` approach.

A third-party module provider who developed a module for the platform must register it before the marketplace provider publishes it. The registration comprises the third-party provider submitting information on the module, e.g., the name and description, pricing options, and contracts and terms. Further, the third-party provider has to add information on technical details describing how consumers can access the module. For example, if the module is accessible via an API, the third-party provider has to provide the API's base URL and the authentication options (OAuth2.0 or API Key). In the next step, the third-party API provider submits the information for review by the marketplace provider. Finally, the marketplace provider sends feedback or approves the service within a set time frame.

However, if the user in C13 decides to become a marketplace and module consumer, she must subscribe to a service by choosing an available monetization plan. With a subscription, the API consumer agrees to the contractual terms of the third-party API provider. After subscribing, if the module is accessible via API, the API consumer can retrieve the API credentials in the marketplace portal and start using it.

Overall, the marketplace described in C13 provides all functionality for third-party API providers and API consumers as a self-service. Nevertheless, many users contact the marketplace and prefer to build personal relationships before publishing or consuming an API.

C14 is a financial services provider that offers software operated in the cloud to end-users. APIs enable other software providers to integrate their software with the system of C14, thus offering an integrated solution to the end users. Consumer organizations have to enter a contractual agreement with the API provider before being able to access the APIs. The process requires non-automated interactions between the API provider and the consumer. However, as soon as a consumer organization has a contractual agreement with the API provider, the consumer can access APIs sandboxes through a developer portal that provides all necessary credentials to authenticate the consumer's application. After the API provider reviews and approves the consumer application developed in the sandbox environment, the consumer receives the credentials to access the production environment. Hence, only the provision of the sandbox API key is a self-service.

Stripe (C16) provides APIs for online payment processing[33]. Consumers that want to use Stripe can choose between two payment options which are a pay-as-you-go model or the option to negotiate an individualized agreement[34]. If a business owner chooses the pay-as-

[33]https://stripe.com/en-de/about (accessed 20.12.2023).
[34]https://stripe.com/de/pricing (accessed 20.12.2023).

you-go model, they can right away register with an email address and password to create a Stripe account[35]. After successful registration and account creation, a consumer can use the API in a test mode[36]. In the test mode, Stripe provides a test API key[37] that enables the business owner to use all Stripe features, but the account can only process test data[38]. However, immediately after registration, the business owner can activate the account by filling in an account application that inquires information about the business, the product, and the relation between the business owner and the business[39]. Stripe collects the information for regulatory reasons[40]. Furthermore, Stripe reviews the information and contacts the consumer in case of questions[41]. As soon as the account is activated, the business owner can access a new API key[42] that allows an application to accept real payments[43]. As a result, most business owners should be able to use Stripes capabilities within minutes[44] and without any manual interaction with Stripe employees.

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[45]. An API consumer has to register to create an account[46]. As part of the registration process, the API consumer must agree to the terms of service, the privacy statement, and other relevant terms[47]. If required, the consumer can upgrade the trial account to a full paid account via a billing page[48]. Hence, Twilio provides complete self-service.

Cross-case observations:

Across all cases, the API initiatives are in production, and the cases capture public and partner API initiatives. Moreover, the API initiatives are developer portals and marketplaces and offer different monetization schemes. Thus, we assume that the API initiatives providing self-services are in more mature stages. Also, the more mature an API initiative, the more onboarding process steps are available as self-service. However, regarding other characteristics, the API initiatives can differ.

[35]https://stripe.com/de/pricing (accessed 20.12.2023).
[36]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[37]https://stripe.com/docs/keys (accessed 20.12.2023).
[38]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[39]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[40]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[41]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[42]https://stripe.com/docs/keys (accessed 20.12.2023).
[43]https://stripe.com/docs/account/manage (accessed 20.12.2023).
[44]https://stripe.com/de/pricing (accessed 20.12.2023).
[45]https://www.twilio.com/ (accessed 20.12.2023).
[46]https://www.twilio.com/docs/usage/tutorials/how-to-use-your-free-trial-account#sign-up-for-your-free-twilio-trial (accessed 20.12.2023).
[47]https://www.twilio.com/try-twilio (accessed 20.12.2023).
[48]https://www.twilio.com/docs/usage/tutorials/how-to-use-your-free-trial-account#how-to-upgrade-your-account (accessed 20.12.2023).

6.7. Integration Partner Program

Pattern Overview	
Name	`Integration Partner Program`
Alias	Integration Partner Management; Integration Partner Referral
Pattern Type	API Consumer-facing Pattern
Summary	API providers support API consumers with finding suitable integration partners by creating and maintaining a curated list of potential integration partners that meet specific quality criteria.

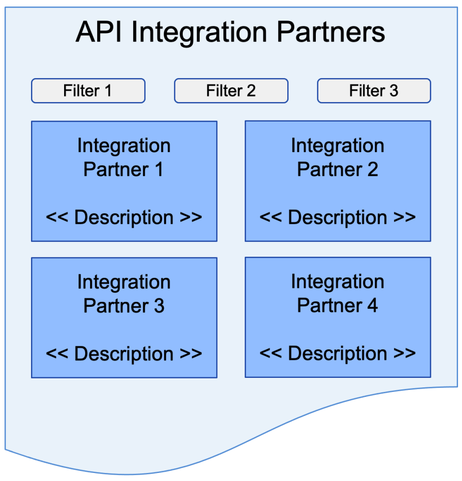

Figure 6.7.: The provider presents a list of integration partners on the API developer portal according to the pattern `Integration Partner Program`.

Context:

API consumers want to integrate a software product into their application landscape via an API. However, not all consumers have the technical capabilities or capacities for such API integration projects. Also, they have not previously worked with a suitable integration or development partner (C4). This context often occurs in settings with an API consumer base consisting of many SME businesses (C14).

Concern:

How can the API providers support API consumers without technical capabilities or capacities to identify suitable API integration partners?

Forces:

- *Lack of integration capabilities and capacities.* Some potential API consumers lack the technical capabilities or capacities to integrate a product into their system landscape.
- *Lack of market expertise.* The potential API consumers do not have an existing relationship with a suitable integration partner (C4). Also, they lack experience in choosing integration partners, which could lead to contracts with integration partners of bad quality. The bad quality of integration partners could negatively reflect on the API provider's product.
- *Lack of budget.* Some potential API consumers lack the budget to hire integration partners to support the integration of a software product into an existing application landscape. For example, this is often a challenge for governmental institutions.
- *Effort.* The API provider has limited resources. Thus, a solution should require little effort.

Solution:

API providers support API consumers with finding suitable integration partners through creating and maintaining a curated list of potential integration partners that meet specific quality criteria. To do so, the API providers attract potential integration partners through existing business relations or via their API portal and validate them according to predefined quality aspects. Integration partners that meet the quality criteria are marketed to potential API consumers via the API portal.

Stakeholders:

This pattern addresses a concern of an *API provider*. The provider has to contact, vet, and gain the interest of *integration partners*. Furthermore, the provider can ask *API consumers* to help identify suitable integration partners.

Implementation Hints:

Partner acquisition. First, the API provider has to identify suitable implementation partners, including software companies and freelancers. Depending on the maturity of the API initiative, approaches to partner acquisition can differ. In the early stages of an API initiative with few API consumers, API providers can simply ask existing API consumers if they employ implementation partners. If so, the API provider can approach the implementation partner with the offer of applying to become an official API provider partner. Another approach is that the API provider collects information on organizations contacting the API management or support team and analyzes if these organizations are implementation partners working for an API consumer. Additionally, the API provider can advertise and explain an implementation partner program on the API portal or at events. Hence, interested implementation partners can contact the API provider, potentially using a dedicated application form (C9).

Quality assurance. An API consumer's bad experience with an official integration partner negatively reflects on the API provider. Hence, the API provider must ensure that officially referenced integration partners meet pre-defined quality criteria. A quality check can target different aspects of the implementation partner organization, e.g., the number of employees with specific competencies, the number of successful past API integration projects, or the general economic status of the organization. Furthermore, API providers can require integration partners to complete API, software, or process training (C14).

Monetization. The API provider can monetize the partner program by charging partners for mandatory training. However, monetization can prevent partners from applying to a partner program. Hence, API initiatives in the early are only beginning to build a partner ecosystem should refrain from monetization (C4).

Tiered partner programs. Advanced integration partner programs can set up a tiered certification process. The integration partner has to achieve specific goals to reach higher certification levels, but a higher certification renders the partner eligible for more benefits, e.g., access to specific resources.

API portal. A dedicated partner list page comprises an overview of all partners. In many cases, the API consumer can apply filters to partner lists, e.g., geographic or industry filters. Furthermore, the partner list can link to separate pages describing each partner, potentially including `Customer Success Stories`, and linking to the external integration partner's website. Finally, the API portal's landing page can refer to the integration partner list for improved visibility.

Consequences:

Benefits:

- *Lack of integration capabilities and capacities.* An integration partner program acts as a broker to match potential API consumers lacking technical capabilities or capacities with integration partners that can realize an integration project.
- *Lack of market expertise.* API consumers profit from integration partner programs since they do not have to search for suitable partners themselves. In addition, since the API provider vets the integration partners, it is less likely that the API consumer will have a negative experience working with the integration partner and the API provider product. Thus, API consumers are more likely to use the API, which leads to additional business for the API provider.

Drawbacks:

- *Lack of budget.* Contracting an integration partner requires a budget.

- *Effort.* The API provider has to design an integration partner program and needs to acquire new and maintain existing integration partner relationships continuously. Also, the API provider has to offer benefits to the integration partners that motivate them to join a partner program, especially if the admission criteria are strict. Such benefits are often access to training material, expert resources, or better support. Hence, the solution can require quite some effort.

Related Patterns within this Pattern Catalog:

The provider publishes the list of partners resulting from the `Integration Partner Program` on the developer portal, thus addressing the information needs of business stakeholders according to `Role-based Marketing`.

In addition, the `Integration Partner Program` enables consumers lacking technical capabilities or capacities to identify suitable API integration partners. Therefore, the program supports the consumers' integration of a `Web API` or `Client Library`.

An alternative to the `Integration Partner Program` is the pattern `Frontend Venture` since both patterns aim to enable consumers lacking technical capabilities or capacities to access `Web API` or `Client Library` functionality. The API provider should create a `Frontend Venture` in a setting where a frontend would meet the need of several API consumers so that it can be evolved into a product. Also, a `Frontend Venture` does not require the API consumer to have any budget for API integration. In comparison, in a setting where a consumer requests an integration into their application landscape and has the budget for such an integration, the API provider should recommend implementation partners to the API consumer.

Known Uses:

We observed the pattern in six cases:

Case C4 represents the public API initiative of a software service provider. End users with a license for the API provider's software product automatically have access to its public API to enable customized integration with their systems. The API provider initiated an integration partner program after realizing that, quite often, software developer firms implementing individual integrations for consumers contracted their support. The goal of the integration partner program is not to generate additional income, e.g., via monetized courses that partners have to complete before becoming an official partner, but to provide end-users of the software with a curated list of software developers that can help them solve their individual challenges and thus increase consumer satisfaction. Instead, the requirements for becoming an integration partner are pretty easy since the provider wants to foster the growth of the integration partner ecosystem. The requirements are that a software development firm has already realized one successful integration project with the provider's API and passes a few formal checks, e.g., has a professional website. As of now, the API provider's developer portal lists all integration partners with links to their respective websites.

In C9, the organization offers a public marketplace for IoT applications. The organization provides core platform software, and third-party developers can build additional modules for the platform using APIs. In addition, consumers can integrate the platform and chosen modules. The API provider created an ecosystem of partners that support customers when working with the platform. One type of partner is the integration partner, who helps customers integrate solutions offered via the platform into their application landscape. Software firms must master a series of courses within a predefined time and implement at least one use case to become official integration partners.

The API provider of C11 offers a public API for easy integration of a financial service into websites and online shops. This organization offers a partner program for different types of partners, including system integrators. In addition to publishing a searchable list of potential integration partners, the API provider also highlights a small number of preferred partners.

Similar to C4, C14 also captures the API of a software product to which users with a software license automatically have access. Access to the API enables the end-users to integrate the software product with their application landscape. The API provider publishes and markets a list of integration partners on the developer portal. The end-users can filter for regions and topics to find integration partners most suitable for them. Also, the API provider requires the integration partner to attend specific courses and acquire certificates before they can become integration partners.

Stripe (C16) provides APIs for online payment processing[49]. Stripe also offers a consulting partner program that allows software firms to become official Stripe consulting partners[50]. These partners are listed on Stripes developer portal[51].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[52]. Twilio offers the partner program *"Build"* for *consulting partners* stating that *"Partners are a vital extension of Twilio's sales organization and go-to-market strategy"* Twilio, 2022, p. 3. One type of consulting partner are *System integrator (SI) partners*. The partner program defines different tiers, which are 'registered,' 'bronze,' 'silver,' 'gold,' and 'global strategic partner.' Consulting partners fall into one of these tiers depending on the number of dedicated certified and trained resources and revenue thresholds. Depending on their respective tier, the consulting partners receive different benefits from Twilio, e.g., training, early beta access to Twilio products, or even market development funds Twilio, 2022.

API consumers can view and filter the list of curated consulting partners on the Twilio developer portal[53].

Cross-case observations:

All these cases have in common that the API initiatives are already in production. However,

[49]https://stripe.com/en-de/about (accessed 20.12.2023).
[50]https://stripe.com/docs/partners (accessed 20.12.2023).
[51]https://stripe.com/partners/directory?t=Consulting (accessed 20.12.2023).
[52]https://www.twilio.com/ (accessed 20.12.2023).
[53]https://showcase.twilio.com/partner-listings?type=Consulting (accessed 20.12.2023).

approaches to integration partner management differ depending on the initiative's maturity and the number of API consumers. More mature API initiatives with a more extensive API consumer base (> 10,000) let interested integration partners approach them and have more complex partner programs, e.g., tiered certifications. API initiatives in the early stages must acquire implementation partners more actively. This makes sense since mature API initiatives are already well connected within their ecosystem, while early-stage API initiatives have to put more effort into establishing such connections.

Also, we observed the pattern `Integration Partner Program` in settings where the API consumers are the end-users who want to integrate a software product into their system landscape.

6.8. API provider-wide ticketing management

Pattern Overview	
Name	`API provider-wide Ticketing Management`
Pattern Type	API Consumer-facing Pattern
Summary	The API provider uses a uniform ticketing system that manages all API-related tickets and is available to all teams involved in API provision. Hence, the ticketing system enables transparency, e.g., on ticket resolution times or recurring issues.

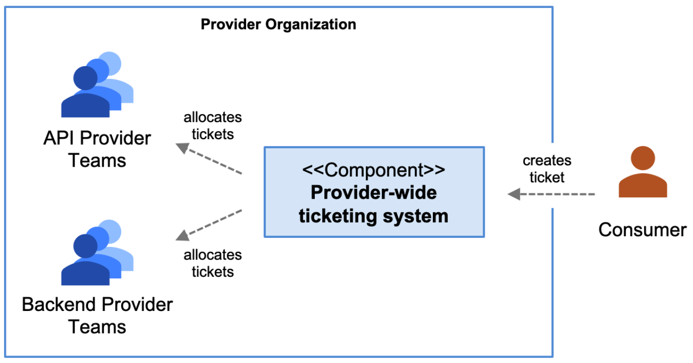

Figure 6.8.: The API consumers enter tickest into the `API provider-wide ticketing management`, which the system routes to the responsible teams within the API provider organization.

Context:

API consumers want to contact an API provider in case of issues or questions. The ability to answer these requests is spread across different teams or roles within the API provider organization (ITIL 4, 2019). Some of these teams or roles are potentially distributed across entities outside the API provider organization, e.g., various subsidiaries (ITIL 4, 2019). These teams or roles should respond to consumer requests in a timely manner not to upset the API consumers. A good support experience can also be a differentiating factor that attracts more developers to an API (Jacobson et al., 2012).

Concern:

How can the API provider transparently and quickly manage consumer requests in an organizational setting with distributed teams in- and outside of the API provider organization?

Forces:

- *Transparency.* In the early days of an API initiative, it is common that API consumers can contact the API provider by email, Slack, or video conferences. However, managing support requests via email prevents transparency. Hence, it is difficult to answer how many support request emails a team or person receives, how long it takes to answer them, or what issues are more common than others. A solution for managing consumer requests should allow for transparency, e.g., the solution should enable the API provider to get an overview of types of tickets, the average ticket resolution time, and recurring issues. Such transparency allows the API provider to uncover improvement potentials (C3).
- *Consumer pressure.* Sometimes a consumer-facing team sits in between the API consumer and the API provider and backend team. Due to a the missing direct contact with the consumer, the backend and API providers do not feel the pressure to resolve API consumer requests with high priority. Thus, a solution should enable direct communication between the API consumer and all teams involved in API provision (C3).
- *High support costs.* An API provider and backend team has limited resources to answer tickets. The costs for supporting different API audiences can differ, e.g., single developers using a public API might ask for more support compared to a strategic partner. An imbalance between the provided support and the business value that each audience group creates can lead to cost-inefficiencies (Jacobson et al., 2012).
- *System introduction.* The introduction of a solution should be of minimum possible effort.
- *Privacy requirements.* Privacy relevant data is part of a ticket, e.g., the API consumer's name or email address. Therefore, the solution must ensure compliance with privacy regulations (C3).

Solution:

An API provider-wide ticketing system holds and organizes all API provision-related tickets. The API consumers or the API provider team itself can create new tickets in the system. All teams involved in API provision, i.e., all backend providers, API providers, and platform providers, have access to the system. The ticketing system allocates the tickets to the responsible teams or roles. Hence, the ticketing system allows for transparency regarding, e.g., ticket resolution times or recurring issues.

Stakeholders:

The different roles within the *API provider* team have to agree on an API provider-wide ticketing system. In this setting, an *API governance team* or a consumer-facing team can lead the decision process. Furthermore, the *upper management* has to approve the introduction of the ticketing system.

Implementation Hints:

Ticket generation and routing. New tickets can enter the ticketing system in different ways. For example, the support staff can generate tickets for issues that API consumers report (C9). Also, API consumers should be able to create tickets themselves via the API developer portal (De, 2017; C3; C9). Finally, new tickets can enter the system automatically, e.g., similar to crash reports in Microsoft (Jacobson et al., 2012).

After a ticket enters the system, it is categorized according to the product it pertains to or the request type, e.g., if it is a technical or business inquiry. Based on the categorization, the ticketing system assigns the ticket to the responsible team or role (C3). All parties involved in API provision must use the same ticketing system, even if the teams belong to different organizations, to ensure that the ticket reaches the team or role responsible for addressing it.

Interaction capabilities. The closer the interaction between API consumers and API providers in case of issues, the better (Jacobson et al., 2012). A relevant capability of the ticketing tool is that the API consumer who generates a ticket can later check the ticket's status (C3). In addition, the API provider should be able to add comments to the ticket, e.g., detailing the measures taken to address the issue. The API provider should also be able to choose whether the API consumer sees these comments. Thus, the API provider can communicate some information to the API consumer and keep other information internal (C3).

Tooling. The API provider has to make a conscious choice for a ticketing system, e.g., JIRA Service Desk[54] (C3) or Gitlab[55] (C2). The ticketing tool should integrate with the organizations API management platform, if one is in place (De, 2017; Spichale, 2017).

When planning to introduce a new ticketing system, the API provider should first check if other teams in their organization already use a ticketing tool. If other teams already use such tools, the API provider should evaluate the tool's suitability for their purposes (C2; C3; C9).

Prioritization. The API provider should define clear rules for prioritizing tickets depending on the importance of the issue or the consumer (Jacobson et al., 2012). For example, a partner using an API might creates the most traffic, but developers using the public API might open many tickets that require much time to answer but add little business value for the API provider. In this case, tickets opened by the partner organization should have a higher priority. Also, the API provider should clearly communicate the expected responsiveness to each audience group, not to set wrong expectations (Jacobson et al., 2012).

Consequences:

Benefits:

[54]https://www.atlassian.com/software/jira/service-management/features/service-desk (accessed 20.12.2023).
[55]https://gitlab.com/gitlab-org/gitlab (accessed 20.12.2023).

- *Transparency.* An API provider-wide ticketing system manages all tickets in one place, and most tools also provide analytics functionality. Thus, API provider-wide ticketing systems allow for transparency and help to reveal improvement potentials. Other communication channels like email, Slack, or video conferences do not provide such transparency and traceability (C3; C7).
- *Consumer pressure.* With an API provider-wide ticketing system, API consumers directly communicate with the internal teams and roles involved in API provision, e.g., via comments and status updates. Thus, all API provider teams feel direct pressure from the API consumers to resolve requests (C3).
- *High support costs.* An API provider-wide ticketing system allows for configuring and enforcing prioritization rules for tickets depending on their audience or the type of issue. Hence, such a tool can prevent cost inefficiencies due to the inadequate allocation of support to different tickets.
- *System introduction.* In case the organization already uses a ticketing tool within the organization, and the ticketing system provides the necessary capabilities, the API provider can introduce the ticketing system with minimum effort. Nevertheless, the API provider organization has to train its employees using the system.
- *Privacy requirements.* Ticketing tools usually provide capabilities for compliance with privacy regulations, e.g., the encryption of personal data or the ability to delete data upon request (C3).

Drawbacks:

- *System introduction.* In case the organization does not already use a ticketing tool, the API provider has to choose and introduce a new ticketing system. The introduction of a ticketing system is expensive and effortful. Licenses for ticketing systems are costly, and the API provider has to configure the system and train its employees (C3).

Related Patterns within this Pattern Catalog:

The `Dedicated Support Team` creates tickets that the `API provider-wide ticketing management` forwards to the responsible teams or persons within the provider organization.

Known Uses:

We observed the pattern in four cases:

The API initiative of C2 makes simulation and modeling algorithms accessible. Internal departments and one partner already access the API, but the API should become public in the future. In this case, the API provider teams adopted the ticketing system already used in the organization.

The API portal provider of an automotive organization (C3) makes vehicle data available to car owners and maintenance data available to workshops. The API provider recently adopted a new ticketing system. Initially, the ticketing system was used only by the API consumer-facing team, but now other teams involved in API provision also adopted the system, even if they belong to other subsidiaries. The chief reason for using a company-wide ticketing system is to move away from emails and to create transparency on the number and types of tickets and the processing times. The support team can enter new tickets into the system, or API consumers can generate tickets using a form on the API portal. The form allows consumers to specify the type of inquiry and the product, adding information to the ticket that allows the system to forward it to the responsible team automatically. Additionally, the chosen tool enables API consumers to check the ticket's status after sending it.

In C9, the organization offers a public marketplace for IoT applications. Again, the API provider teams adopted the ticketing system already in use within the organization.

C14 is a financial services provider that offers software operated in a cloud to end-users. APIs enable other software providers to integrate their software with the system of C14, thus offering an integrated solution to the end-users. In this setting, the API providers also adopted the ticketing system already in use in the organization. However, the system first sends all tickets to a dedicated consumer-facing team, which assigns the tickets to the responsible parties.

Cross-case observations:

The cases are very heterogeneous, as they span public and partner API initiatives with low to high numbers of API consumers. Also, the API initiatives are developer portals as well as marketplaces and monetization approaches cover primarily contractual agreements and free API access as part of a software license purchase, but also free and pay-per-use plans. Three of the API initiatives are in production, and one is still in the pilot phase. Further, all API initiatives focus on B2B consumer relationships, with one API initiative operating in the B2C and B2G sectors. Thus, the application of the pattern `API provider-wide ticketing management` seems to be beneficial to a broad range of API initiatives.

6.9. Dedicated Support Team

Pattern Overview	
Name	`Dedicated Support Team`
Pattern Type	API Consumer-facing Pattern
Summary	The dedicated support team accepts all API consumers' questions, service requests, and incident reports and immediately answers or resolves low- or medium-complexity tickets. Only high-complexity tickets are forwarded to the respective experts, relieving the API and backend provider teams of a portion of the support activities.

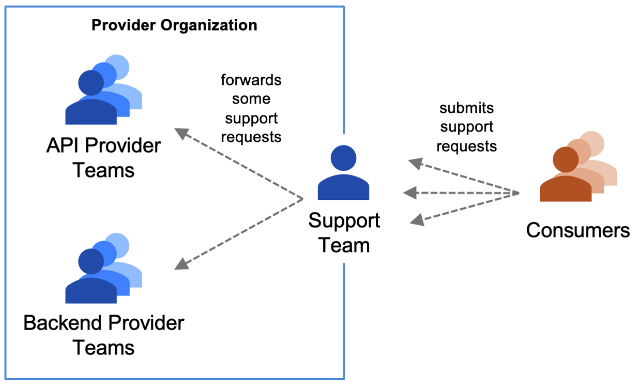

Figure 6.9.: The `Dedicated Support Team` forms the interface between the API consumer and provider. The support team solves low- or medium-complexity tickets and forwards more complex requests to the respective experts in the provider organization.

Context:

An API provider designs, develops, tests, and publishes an API to external users. However, the satisfaction and user experience of consumers depend not only on the API itself but also on the resources and processes that support the API consumer with the use of the API (ITIL 4, 2019; Jacobson et al., 2012; Medjaoui et al., 2018). Hence, the API provider publishes resources that help API consumers to understand and use the API, e.g., documentation like `Consumer-centric API Description` or `Integration Guide`. Nevertheless, the additional resources cannot answer all consumer questions. Furthermore, the API consumer sometimes runs into issues with the API itself, e.g., if the API is down (Jacobson et al., 2012). In such events, API consumers expect to be able to contact the API provider who will solve their problems so they can keep being productive (De, 2017; ITIL 4, 2019).

Concern:

How can an API provider support an API consumer in case of questions, service requests[56], and incidents[57]?

Forces:

- *Support as success factor.* Support is a success factor for API initiatives (Medjaoui et al., 2018), and API providers can differentiate their APIs from competitors by offering better support services (Jacobson et al., 2012).
- *Resource specialization.* The API provider teams have many different tasks, including the operation and evolution of APIs. A solution should minimize the support activities for API provider teams to free up resources for more complex tasks that advance the API initiative (C14).
- *Immediate resolution.* API consumers requesting support do not like to be forwarded or to hang up and wait for a return call from the API provider organization. Thus, a solution should allow API consumers to talk to an entity that can resolve their questions and issues right away (C14).
- *Scaling.* In the case of a successful API initiative, the number of API consumers increases over time. With an increasing number of consumers, it is likely that the number of support requests increases as well (Jacobson et al., 2012).
- *Personal connections.* The creation of a personal relationship between members of the API consumer and the API provider organizations increases the likelihood of a long-term business relationship (Medjaoui et al., 2018).
- *Internal collaboration.* The resolution of complex incidents can require the collaboration between different stakeholders of an organization, e.g., API provider and backend provider teams (ITIL 4, 2019). A solution should not add any unnecessary coordination and communication levels and efforts.
- *Morale.* A solution should foster the internal morale and motivation of all employees of the API provider organization.
- *Cost.* Additional support leads to increased costs. The API provider has to decide how much funding they want to invest in support activities (Jacobson et al., 2012).

[56]ITIL defines a service request as *"A request from a user or a user's authorized representative that initiates a service action which has been agreed as a normal part of service delivery."* (ITIL 4, 2019, p. 156).

[57]ITIL defines an incident as *"An unplanned interruption to a service or reduction in the quality of a service."* (ITIL 4, 2019, p. 121). Problems cause incidents, and a service provider needs to resolve a problem or find a workaround to reduce future incidents (ITIL 4, 2019).

Solution:

The API provider has a dedicated support team as the first point of contact for API consumers. The dedicated support team accepts all API consumers' questions, service requests, and incident reports. The support team immediately answers or resolves low- or medium-complexity questions, service requests, and incidents. However, in case of high-complexity questions, service requests, and incidents, the support team forwards these support requests to experts in the API provider and backend provider teams. Since the support team relieves the API provider and backend provider teams of a portion of the support activities, the API provider and backend teams can focus on support activities requiring their expertise and other API management tasks, e.g., API evolution.

Stakeholders:

This pattern addresses a concern of an *API provider*. The provider has to onboard a potentially already existing *customer support* team and continuously transfer knowledge regarding changes of the API to them. Additionally, the support team has to report statistics and insights on consumer support requests and incidents back to the API provider to support the improvement of the API initiative, e.g., if consumers repetitively ask for clarification due to ambiguous documentation. The API provider potentially also has to collaborate with the *backend providers* to answer complex support requests or to resolve specific incidents. Furthermore, the formation or onboarding of a dedicated support team constitutes a change of the organizational structure. Therefore, the API provider needs the approval of the *upper management*.

Implementation Hints:

Process. The support team is the first point of contact for API consumers, e.g., if an API consumer wants to report technical issues, make a feature request, or ask questions regarding the functionality of an API (De, 2017; C5). The API provider organization can offer API consumers different means to contact the support team, e.g., via email, phone, or chat function (Medjaoui et al., 2018; C5). The API consumer can find the contact information or other means to contact the support team on the developer portal (De, 2017; Jacobson et al., 2012; C5). As soon as an API consumer contacts the support team with a question, service request, or incident report, the support team must categorize the request into one of two categories. The first category comprises support requests and incidents that the support team can answer or resolve immediately. The second category comprises support requests and incidents the support team can not answer and thus needs to forward to the responsible API or backend providers (C4; C5; C14). The responsible API or backend team then takes over the communication with the API consumer and answers the request or resolves the incident or problem (C4; C5). As a result, the API provider and backend provider teams have to manage fewer support requests and incidents and can focus on development activities instead (C14).

In addition, it is also crucial that the support team collects and communicates repetitive support activities back to the API provider and backend provider teams. This kind of feedback allows the API provider and backend provider teams to improve the API initiative, e.g., by removing ambiguities in the documentation. Such improvements help to reduce the number of support requests and incidents altogether (C4).

Organizational structure. The support team is organizationally independent of the API provider and backend provider teams (C5). In the cases that form the basis of this pattern catalog, all API provider organizations already had dedicated support teams for the product or platform to which the API belongs. These support teams additionally took on the support for the API channel.

Division of responsibility. We observed that the organizations always have two tiers of support (Walker, 2001), which are a dedicated support team as the first tier and API provider and backend provider teams as the second tier.

With two tiers, the API provider organization has to decide which types of support activities the first tier should handle. In some cases, the support team might only answer routine questions or refer the API consumer to relevant information sources, e.g., FAQs. Such routine inquiries can, for example, cover whether an API exists for a specific software or what high-level functionality an API provides (C5). However, as soon as the support activities are more technical or require non-public knowledge, the support team forwards the request and incidents to the respective API provider or backend teams (C5; C14).

In other cases, the support team additionally resolves specific technical and slightly more complex requests and incidents. Hence, the support forwards fewer support requests and incident reports to the API provider and backend providers (C14). However, this approach increases the need for training in the support team and precise documentation.

Knowledge transfer. The API provider has to enable the support team to respond to support requests and incidents. Thus, the API and backend provider teams must transfer knowledge to the support team. An option is that the API provider team creates an internal FAQ document that guides the support team. The support team can use a `Growing FAQ` which they continuously extend and adapt to cover unforeseen questions (C4).

Expectation management. It is essential to clearly communicate the expected support response and resolution times to the API consumers (Jacobson et al., 2012). API consumers expect responses to support requests immediately, even if the API provider only informs the API consumer that they are working on finding a solution (Jacobson et al., 2012).

Ticket management. The provider organization can use a `API provider-wide ticketing management` to create and forward tickets and manage communication between internal teams and the API consumer (C14).

Consequences:

Benefits:

- *Support as success factor.* A dedicated support team can improve the support services and thus contribute to the success of an API initiative.

- *Resource specialization.* Since the support team takes over a part of the support activities, API provider teams can spend more time on tasks that advance the API initiative, e.g., API design and evolution activities (C14). Thus, a dedicated support team shields expensive experts from having to process non-complex support requests and enables optimized resource allocation (Walker, 2001). However, a dedicated support team cannot completely free up the API experts from all support activities, e.g., in case of complex incidents (ITIL 4, 2019).
- *Immediate resolution.* The API provider team has limited resources. Therefore, in case of a high volume of API consumer requests, the API provider might not answer requests immediately and has to call the API consumer back later. Especially if the API consumer has a small and simple question, waiting for a return call negatively impacts the consumer experience. Thus, a dedicated support team that answers a portion of support requests immediately can improve the consumer experience.
- *Scaling.* The API provider has to decide on strategies to scale the support for a growing API consumer community (Jacobson et al., 2012). By distributing support requests across the support, API, and backend provider teams, the API provider organization can answer more support requests overall.

Drawbacks:

- *Resource specialization.* A dedicated support team can free up expert resources for other activities, but it is not the only possible approach to reducing support requests to experts. Alternatively, the API provider can publish high-quality documentation or provide other self-help facilities. For example, the API provider can provide a public website where API consumers can post questions that the API provider team answers or create a community that enables API consumers to help each other (Jacobson et al., 2012). Such a measure should reduce the number of repetitive, low-complexity support requests to the API provider organization in the first place.
- *Immediate resolution.* The support team is the first point of contact for API consumers. However, the API provider team has the most expertise regarding the API, and in case of complex requests, the support team has to forward the ticket or call to the API provider team. Depending on the API provider's processes, the API provider team might even call API consumers back later. Also, API consumers might have to explain the support request several times to different API provider support members (Walker, 2001). Thus, using a dedicated support team might negatively impact the consumer experience (C14).
- *Scaling.* Even if a dedicated support team adds additional resources for answering support requests, each support team member can only take over a limited amount of API consumer requests. Thus, the support team needs to grow with an increasing

number of API consumer requests. Moreover, the support team can only take over the portion of support requests within their abilities and assigned responsibilities. Thus, a support team can not relieve the API provider team of an increasing number of higher complexity issues.

- *Personal connections.* A dedicated support team does not assign single members to support specific API consumers. Also, the support staff can only solve a specific set of support requests and has to forward the consumers' requests to the API or backend provider teams in some cases. Thus, a dedicated support team does not foster long-term personal relationships with members of the API consumer organization.
- *Internal collaboration.* A dedicated support team as the first point of contact for API consumers adds another party to the support request resolution process. For example, the dedicated support team has to collect initial information about an incident and communicate it to other teams. Similarly, after the resolution of a problem, potential API changes might have to be communicated to the support team (ITIL 4, 2019). Hence, a dedicated support team increases internal coordination and communication efforts for the API provider organization (Walker, 2001).
- *Morale.* A dedicated support team can lead to members of the support team being seen as lower-class employees, which might negatively impact morale and motivation within the organization (Walker, 2001).
- *Cost.* A dedicated support team increases the cost of support activities.

Related Patterns within this Pattern Catalog:

Consumers can report issues with an interface to the `Dedicated Support Team`, which can solve low to medium-complexity support requests immediately. In addition, the `Dedicated Support Team` can create tickets for higher complexity issues that the `API provider-wide ticketing management` forwards to the responsible teams or persons within the provider organization.

Also, the API provider and backend teams have to enable the support to answer support requests and solve incidents. A pattern candidate that allows for knowledge transfer and creation is the `Growing FAQ`.

Known Uses:

We observed the pattern in four cases:

The API provider of C4 provides a financial services software product. In addition, the API provider offers an API that allows users of the software product to integrate the software product with their system landscape. The software product has an integrated tool that allows users to create tickets. The tool sends the tickets to the software's support team, which answers or forwards inquiries regarding the associated APIs. The organization uses internal

FAQs to transfer knowledge from the API provider team to the support team. The support team adapts and extends the FAQ over time to meet the support's needs. Additionally, the support team communicates repeating questions to the API provider team to enable the API provider team to improve the APIs and decrease the support request volume.

C5 is a different API initiative of the same organization described in C4. Again, the starting point is a financial services software product, but the APIs aim to enable other software providers to integrate their software with the product, thus offering an integrated solution to the end users. A dedicated support team receives all support requests regarding the APIs via email, phone, or chat. The support team answers high-level questions regarding the API right away or refers the API consumer to resources, e.g., FAQs and documentation, that solve their problems. However, the support team forwards technical requests to the API provider team.

In C9, the organization offers a public marketplace for IoT applications. The organization provides core platform software, and third-party developers can build additional modules for the platform using APIs. The API provider has a dedicated support team that answers support requests from third-party developers.

C14 is a financial services provider that offers software operated in the cloud to end-users. APIs enable other software providers to integrate their software with the system of C14, thus offering an integrated solution to the end users. The API provider organization has a dedicated support team that is the first point of contact for all consumer inquiries. The support team can answer high-level questions regarding the API but does not solve any technical issues. Instead, the support team forwards these tickets to the responsible API team. In the future, the API provider organization wants to enable the support team to solve simple technical issues to reduce the number of support requests for the API teams. Furthermore, the immediate resolution of issues also improves the consumer experience.

Cross-case observations:

All API initiatives have in common that they are in production and focus on a B2B audience. Furthermore, the cases comprise public and partner API initiatives, of which three are developer portals, and one is a marketplace. However, monetization differs between the cases and includes contractual plans, free use, and free use as part of a product license.

Most interesting, however, is that all API initiatives belong to existing software products or platforms. For these software products and platforms, dedicated support teams already existed. Hence, the existing support teams also took over the role of a dedicated support team for high-level support requests and incidents for the API initiative.

6.10. Service-Level Agreement (SLA)

Pattern Overview	
Name	`Service Level Agreement (SLA)`
Pattern Type	API Consumer-facing Pattern
Summary	An SLA is an agreement between two parties that specifies the quality of services, i.e., the APIs' non-functional properties and support service levels, as well as contractual punishments in case of SLA breaches (Zimmermann et al., 2022). Hence, an SLA increases API consumers' trust in an API's quality.

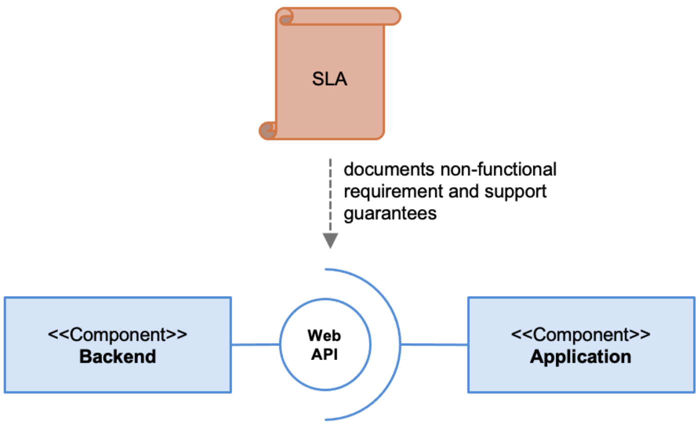
SLA
documents non-functional requirement and support guarantees
<<Component>> Backend
Web API
<<Component>> Application

Figure 6.10.: An `Service-Level Agreement (SLA)` specifies the quality of services for an API.

Context:

API consumers rely on an external API's functionality or data to realize their business model (Zimmermann et al., 2022). Downtimes or unsatisfactory performance of the API can negatively impact the API consumers' business (Zimmermann et al., 2022). Thus, API consumers want guarantees that APIs meet specific non-functional requirements over time and that the API provider treats their interests with sufficient priority before choosing an API.

Concern:

How can API providers identify API consumers' non-functional requirements and create the trust that the API will meet these non-functional requirements?

Forces:

- *Trust.* It is difficult for an API consumer to anticipate the quality of an API over time.
- *Alignment of interest.* The backend provider, API provider, and API platform provider can belong to the same or different organizations. Either way, some teams might have different interests, e.g., the backend provider prioritizes its own tickets over tickets created by the API provider (C3).
- *Special Requirements.* API consumers often want to negotiate special conditions for their endeavors, even if a standard contract exists (C13).
- *Contractual risk.* Even if the API provider takes adequate measures to meet specific non-functional requirements, unexpected downtimes or other issues may occur.

Solution:

Service-level agreements (SLAs) are agreements between two parties defining target values for non-functional properties of APIs and measures in case the APIs do not meet these properties (Zimmermann et al., 2022). An SLA can also include prescriptions on issue management processes and support services. Thus, the API consumer can define measures that motivate the API provider to provide a certain API quality. At the same time, the API provider can monetize APIs with consistently high quality better.

Stakeholders:

The *API provider* negotiates the contents of an SLA with the *backend provider* and the *API consumer*. Furthermore, *legal* supports SLA negotiations and potentially creates a standard SLA contract (Zimmermann et al., 2022).

Implementation Details:

SLA contents. An SLA defines service levels comprising non-functional properties that an API has to achieve at all times and issue resolution or support services in case the API fails to meet these properties (Jacobson et al., 2012; Zimmermann et al., 2022; C6). The non-functional properties are, for example, maximum downtimes or maximum data loss (C6). As part of an issue resolution process, an SLA can define, e.g., at what times the API consumer can reach the support, or the maximum length for the API provider to respond to a consumer request (C3; C6). In addition, SLAs can define maintenance and downtime information (De, 2017). Also, additional provisions concerning the collaboration between API provider and API consumer can be part of the SLA, e.g., how long an API will be in operation without changes or how much time in advance changes have to be announced (Zimmermann et al., 2022; C12).

Which service level an API consumer chooses should depend on the use case that the consumer wants to realize (C6). The breach of an SLA can be tied to monetary penalties to motivate the parties to adhere to the guarantees (Zimmermann et al., 2022; C3).

SLAs and other contracts. An SLA can be part of a more comprehensive contract that also includes provisions on monetization, a maximum number of allowed transactions, or restrictions to the use of the accessible functionality or data (C6).

Types of SLAs. SLAs should be negotiated between all parties involved in API provision and consumption. However, during the case analysis, three types of SLAs stood out in particular, which are:

- *Backend SLAs.* First, in the most simple setting, an API provider provides APIs which consumers use. In this setting, the API provider and the API consumer directly negotiate an SLA.
- *Marketplace SLAs.* Second, in marketplace settings, the API platform provider has to manage API and backend providers from different organizations as well as API consumers. In this setting, the API platform provider can ensure the quality of the services on the marketplace by enforcing service levels with the API providers according to API consumer requirements (C6).
- *Gateway SLAs.* Finally, the API platform is part of the infrastructure for API provision. Thus, the non-functional properties of all APIs provided on the platform depend on the properties of the API platform. Therefore, the API platform has to meet the SLA of the API with the most demanding service level. The API platform provider can make these service levels explicit in an SLA to the backend providers (C6).

Generally, the API provider has to consider the performance of all services that the API depends on and potentially make SLAs with them to ensure that the API can meet its SLA (Jacobson et al., 2012).

Public and partner APIs. Furthermore, SLAs often differ between public and partner APIs. SLAs for public APIs are usually relatively weak, meaning that the API providers do not want to make any firm promises, even if they internally strive for good performance. In partner API constellations, SLAs are often more robust. In either case, it is essential to communicate if an SLA is in place and what the SLA covers (Jacobson et al., 2012; Zimmermann et al., 2022). For public APIs, the API provider should publish all relevant information about SLAs on the developer portal (De, 2017).

Standard SLAs. The API provider can predefine standard service levels and a standard SLA agreement, which the involved parties can adopt without changes or adapt to their specific needs (C13). Also, during SLA negotiations or standard SLA creation, the API provider may involve the *legal* department (C13).

Monitoring. The API provider needs to monitor the service level, which is a capability that an API platform should be able to provide (De, 2017). The API gateway intercepts all API calls and can apply the pattern `API Quality Monitoring` to audit the compliance with the non-functional requirements defined in the SLA (C4; C5; C13).

Infrastructure and Processes. The backend provider, API provider, and API platform provider need to adapt the infrastructure that enables the provision of APIs depending on the respective service levels. Also, cloud infrastructures allow for easier redundancies and automation in case of downtimes (C6).

Consequences:

Benefits:

- *Trust.* A binding agreement on the quality of an API can increase the API consumers' trust in an API initiative (Zimmermann et al., 2022). Also, the availability of publicly accessible standard SLAs influences an API platform's external perception positively (C3).
- *Alignment of interest.* SLAs create incentives that align the interests of all parties involved in API provision (C3).

Drawbacks:

- *Special Requirements.* SLA negotiations can be laborious and lengthy. API providers can rarely use standard contracts without modification since API consumers want special conditions. Also, the API provider usually has to involve the legal department (Zimmermann et al., 2022; C13).
- *Contractual risk.* Unexpected downtimes or other issues may occur even if the API provider took adequate measures. Since SLAs can trigger monetary penalties (Jacobson et al., 2012), such events can be costly.

Related Patterns within this Pattern Catalog:

An `Service-Level Agreement (SLA)` defines thresholds for performance metrics. Since `API Quality Monitoring` entails the continuous testing of the non-functional properties of an API, it allows the provider to identify breaches of these thresholds (Jacobson et al., 2012; C4; C5).

In addition, the `Testing Strategy` tests the non-functional properties of a new or changed API. Thus, the `Service-Level Agreement (SLA)` also defines non-functional testing goals for the `Testing Strategy`.

Other Related Patterns:

Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018) introduce the pattern `Service Level Agreement`, which, in essence, captures the same solution as this pattern. The authors describe an SLA as the explicit and unambiguous definitions of the *Quality-of-Service* (QoS), the service support, the monitoring processes, and the consequences for not meeting these prescriptions. An SLA contains measurable *Service Level Objectives* (SLOs) to realize this goal. An SLO measures, for example, performance, scalability, or availability. The API provider has to define a threshold value, the unit of measurement, the monitoring conditions, and the interpretation for each SLO. Thus, SLAs should contain explicitly defined metrics instead of ambiguous free-form text. Also, Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018) also recognize that SLAs are especially relevant in settings in which the API is business-critical for the API consumer. Also, the authors agree that SLA design and maintenance are effortful since the API provider has to involve many internal and external stakeholders in SLA negotiations. Moreover, monitoring and risk mitigation efforts can be costly. Thus, many public APIs publish weak or no SLAs. Finally, Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018) introduce an internal SLA as an additional SLA variant. An internal SLA specifies QoS and other measures internally but does not propagate this information to external stakeholders.

Hence, the essence of the pattern `Service Level Agreement` presented in Zimmermann et al. (2022), Zimmermann et al. (n.d.), and Stocker et al. (2018) and the pattern `Service-Level Agreement (SLA)` presented in this pattern catalog is very similar, but the scope and level of detail differ. Therefore, we view the two pattern descriptions as confirmation and extension of each other.

Known Uses:

We observed the pattern in six cases:

In C6, the API provider is a subsidiary within a mobility group that offers API management services. The backend providers and API consumers are both subsidiaries within the same group. Inside the group, a set of predefined service levels focusing on maximum downtimes and support exist. For example, the highest service level ensures the availability of a responsible person 24 hours a day and seven days a week plus immediate issue resolution in case of a downtime breaching an SLA provision. Lower service levels only ensure availability of support during business hours and an issue resolution time of one day. API providers have to choose which service level to adopt depending on the business-criticality of the APIs functionality. Since all APIs rely on the infrastructure provided by the API platform, the API platform has the highest service level. The SLAs also include information on incident management processes but do not include monetary penalties in case of SLA breaches.

The API provider of C10 offers a financial service for API consumers to integrate into their systems. Smaller API consumer organizations have to accept a standard SLA. Since the API provider offers a financial service with high business criticality, the availability thresholds of the SLA are high. However, larger API consumers can also negotiate special conditions, including increased support services.

C12 is a financial services provider that provides software to end-users. APIs enable other software providers to integrate their software with the system of C12, thus offering an integrated solution to the end-users. The API provider teams of C12 negotiate SLAs with other software providers. The SLAs also include information concerning the timeframe during which the API provider does not change an API and how long before a change happens it is announced.

Next, in C13, the organization offers a public marketplace for financial services applications. The API marketplace provider offers a core platform software, and third-party developers can build additional modules for the platform using APIs. Users of the platform can buy the core platform software and add pre-integrated modules according to their needs. The API provider team negotiates SLAs with third-party developers providing modules and with users of the platform. A standard SLA exists, but the partners usually want to negotiate special conditions. Self-service capabilities for users do exist, but in a limited fashion, since it requires high effort to create transparent SLAs and monetization schema. The API marketplace gateway acts as a central point of truth for monitoring the non-functional properties of the platform and its modules and reports issues to the responsible parties.

Stripe (C16) provides APIs for online payment processing[58]. Stripe makes different support plans available to API consumers[59]. However, to the best of the authors' knowledge, Stripe does not assert uptimes or compliance with other non-functional metrics in their SLA[60].

Twilio (C17) is an API provider for customer engagement using voice, messaging, video, and email services[61]. Twilio presents predefined support plans, containing, e.g., different guaranteed support response times or different communication channels to reach the support staff[62]. In addition, Twilio publishes an SLA that defines monthly uptime percentage thresholds for different service levels and fines in case of breaches [63]. Furthermore, the SLA describes how API consumers can claim these fines.

Cross-case observations:

The cases span partner, group, or public API initiatives and are single API portals offering only the organization's APIs or API marketplaces. However, except for one initiative, all cases are already in production. Additionally, all API initiatives focus on B2B settings. Furthermore, we observed a significant difference between the group of cases spanning Twilio, Stripe, and case C10 and the API initiatives C6, C12, and C13. The API initiatives of Twilio, Stripe, and C10 are very mature, with APIs being their primary product distribution channel and many API consumers using the APIs. In these cases, new API consumers have to accept the predefined SLAs. The only exceptions are very big organizations that can negotiate new conditions. On the other hand, the API initiatives C6, C12, and C13 have relatively small amounts of API consumers and report that they negotiate new SLAs with most new partners.

[58]https://stripe.com/en-de/about (accessed 20.12.2023).
[59]https://stripe.com/de/support-and-services#compare-plans (accessed 20.12.2023).
[60]https://stripe.com/en-de/legal (accessed 20.12.2023).
[61]https://www.twilio.com/ (accessed 20.12.2023).
[62]https://www.twilio.com/support-plans (accessed 20.12.2023).
[63]https://www.twilio.com/legal/service-level-agreement (accessed 20.12.2023).

This makes sense since new consumers can influence the API initiatives tremendously and thus have more negotiation power.

CHAPTER 7

Pattern Candidates

All identified solution approaches derived from the case base are pattern candidates at first. However, only if we observed the successful implementation of the solution approach in three cases, thus fulfilling the *rule of three* (Coplien, 1996), a pattern candidate becomes a pattern. Therefore, this section summarizes the API management pattern candidates we observed in less than three cases.

This section presents summaries of the pattern candidates previously published in Landgraf (2021) and of additional pattern candidates identified during the second data analysis. The pattern candidates are categorized into *Interface Type Pattern Candidates*, *API Provider Internal Pattern Candidates*, and *API Consumer-facing Pattern Candidates*.

7.1. Interface Type Pattern Candidates

Open-source SDK

The API provider publishes open-source software libraries and related documentation material to enable API consumers to contribute to the design and maintenance of these resources.
Known uses: C11

Plug-ins

The API provider can implement plug-ins for popular software ecosystems, e.g., Wordpress[1], on top of the APIs. The plug-ins allow consumers to access the functionality and data provided by the API.
Known uses: C4

[1]https://de.wordpress.org/plugins/ (accessed 20.12.2023).

7.2. API Provider Internal Pattern Candidates

Service Feasibility Workshops

The API provider organizes workshops to evaluate the feasibility of APIs with backend provider teams. These workshops increase trust and commitment between participants.
Known uses: C6

Service Creation Guideline

Sometimes it is not clear for an organization which services to offer as a public, partner, or group API. Therefore, the API provider creates a service creation guideline that documents the criteria that exclude services from being published as APIs.
Known uses: C6

Data Clearing Office

This pattern candidate specializes the pattern `API Clearing Process`. The data clearing process ensures that all API endpoints comply with legal and strategic requirements before they are published externally by involving different stakeholders who provide feedback and need to sign off on a new API or a majot change to an existing API. In addition, the API provider can implement a clearing office, i.e., an interdisciplinary committee that needs to approve new or changed APIs, to centralize the data clearing effort.
Known uses: C3

Internal Tech Talks

The API provider organizes tech talks, roadmap presentations, or other internal events to promote and increase awareness of APIs within the API provider organization.
Known uses: C4, C5

Intranet and Internal Social Media

The API provider uses the intranet and internal social media platforms to promote and increase awareness of APIs.
Known uses: C6

Support Hero

The support hero is a role that is assigned to is API provider team members in a rotating manner. The team member with the assigned support hero role is responsible for solving incoming consumer requests during that time.
Known uses: C4, C5

Quarterly Alignment Meetings

All members of the API provider team and the backend provider teams meet quarterly to define goals for the next quarter. These meetings strengthen the commitment of all teams and allow the API team to convince the backend providers to implement the required functionality.
Known uses: C4, C5

Penetration Testing

The API provider tests the API gateway for vulnerabilities that allow adversaries to exploit the system. Penetration testing can be done in-house, or the API provider can employ third-party security firms to provide these services.
Known uses: C6

Several Developer Portal Instances

In a group setting, an API provider can allow different organizational units to create separate access configurations for their APIs by allowing these subsidiaries to instantiate their own developer portals. Each subsidiary with an API developer portal instance can configure APIs' access and visibility according to their needs, thus creating custom views on the APIs. At the same time, the APIs run on the same infrastructure, i.e., API gateway, and can be managed consistently.
Known uses: C6

Tenant Isolation

If the API consumers send data to the API for processing, the API provider needs to ensure that the API does not mix up or expose data of different API consumers to each other. Hence, the API provider uses tenant isolation management techniques to prevent such errors. This pattern candidate is similar to the pattern `Single Tenancy` (Newman, n.d.).
Known uses: C7, C9

Data Anonymization

In some cases, the API provider has to ensure that members of the API provider organization cannot view sensitive data that the API consumer sends to the API, e.g., for privacy or antitrust reasons. Thus, the API provider makes an anonymization tool available to the API consumer that allows the API consumer to anonymize the data before sending it to the API.
Known uses: C7

APItect

An APItect is a specialist who consults internal teams on API design.

Known uses: C14

7.3. API Consumer-facing Pattern Candidates

Contact Form Automation

The API portal predefines ticket categories (e.g., "technical support," "request demo," or "contact sales") which the consumer has to select when contacting the API provider. Based on the ticket type selection, the ticket is automatically forwarded to the right contact within the API provider organization.

Known uses: C3

Smart Contact Form

The API portal predefines ticket categories (e.g., "technical support," "request demo," or "contact sales") which the consumer has to select when contacting the API provider. Based on the choice of the ticket category, the contact form requests different information from the API consumer. For example, a technical support request can ask the API consumer to provide log files.

Known uses: C3

Account Management

The API provider can assign dedicated support teams to strategic partners, comprising technical and business support experts.

Known uses: C6, C10

Procurement Integration

The API consumer might involves procurement in the process of buying access to an API. If this is the case often, the API provider can offer functionality targeted at procurement, e.g., SAP integrations.

Known uses: C13

Keyword Marketing

The API provider can use keyword-based marketing to increase the discoverability of APIs.

Known uses: C4

Pilot Workshops

API providers organize pilot workshops with API consumers to kick off new pilot projects. These workshops aim to collect initial feedback and requirements from potential API consumers and establish trust through personal relationships.
Known uses: C6, C12

Conferences Talks

API providers promote APIs at conferences to increase their discoverability.
Known uses: C4

Bar Camps

Bar camps are conferences the API provider organizes without scheduled talks to collect consumer feedback and enable networking.
Known uses: C12

Support Community

The API provider operates an online forum enabling API consumers to exchange experiences and support each other (De, 2017).
Known uses: C3

API Status Page

The API gateway team can use an API status page to automatically report issues and defects of backend services and API platforms. The API status page should be displayed on the developer portal to inform API consumers of current downtimes or issues. Automated publishing of this information can reduce the volume of incoming redundant bug reports.
Known uses: C3, C13

Changelogs

API providers publish changelogs for their APIs on their API portal to enable API consumers to understand the types of changes and change frequency.
Known uses: C4

Growing FAQ

The API provider continuously updates the externally published or internally used FAQ with common questions and their answers.
Known uses: C4

Sample Projects

The API provider publishes open-source integration examples, i.e., simple API clients implementing common use cases of the API. Also, the API provider needs to publish documentation for these sample projects.
Known uses: C11

API Test Cases

API providers offer test cases to allow API consumers to test their implementations in a sandbox environment.
Known uses: C10, C11

Blogs

API providers should publish articles highlighting different aspects of their APIs, `Customer Success Stories`, and other supporting and promotional information on a blog.
Known uses: C3, C11

Video Series

The API provider publishes videos demonstrating the implementation of common API use cases as part of the API documentation (Jacobson et al., 2012).
Known uses: C11

Business Functionality Description

In some cases, the API provides functionality for complicated business processes, e.g., taxation of business transactions. In such cases, the API provider should publish descriptions that enable technical experts to understand the business functionality/domain.
Known uses: C4

Social Media

The API provider uses social media to promote APIs and communicate with API consumers.
Known uses: C3

Third-party API Quality Assurance

The API marketplace provider wants to ensure the quality of third-party APIs offered for a platform. Hence, the API provider must define a selection and onboarding process that assures third-party APIs meet specified quality criteria.
Known uses: C13

Third-party API Quality Monitoring

An API marketplace provider should continuously monitor the quality of third-party APIs provided for a platform by defining and monitoring KPIs. The API marketplace provider can use an API gateway to automate the monitoring and notify the API marketplace provider team in case of quality breaches.
Known uses: C13

White-label Marketplace

An API provider can generate profits from proprietary API marketplace software by selling the marketplace software component to other organizations as a customizable product.
Known uses: C13

Role System in Developer Portal

The developer portal offers features and views tailored to different user roles, e.g., technical vs. business stakeholders. Furthermore, the definition of different roles enables role-based authorization.
Known uses: C13

Group Access

An API provider can allow API consumers to define different roles for accessing the same consumer account.
Known uses: C9

CHAPTER 8

Summary

APIs are strategic resources (Yoo et al., 2010) at the interface between functionality providers and consumers. Thus, this pattern catalog aims to identify and codify proven API management practices focusing on collaboration between stakeholders for different kinds of organizations, including established organizations. We conducted 16 expert interviews and enriched the data with information on public API initiatives, existing pattern languages and catalogs, and practice-driven API management literature. As a result, we present 22 patterns and their relations, as well as 37 pattern candidates.

Furthermore, we observed the following four general findings during the creation of the API management pattern catalog (previously published in Bondel et al. (2022):

(1) ***"Most initial collaboration between the API provider and the API consumer happens through software artifacts controlled by the API management team"*** **(Bondel et al., 2022).**

API consumers search and inform themselves about interesting APIs online, i.e., via the API developer portal. Moreover, they use the features of the developer portal, e.g., contact forms, to contact the API provider. Also, the consumers use the developer portals `Onboarding Self-service` capabilities to test and use the API. Thus, the successful collaboration between API management and API consumers depends heavily on resources the API management team controls.

(2) ***"API consumers want personal contact with the API provider before and during integrating an API"*** **(Bondel et al., 2022).**

Even if API consumers discover an API via its developer portal, personal contact with the API provider is often a deciding factor for integration. Especially if the API provides domain-specific functionality instead of commodity functionality, API consumers want to negotiate individual contracts, including agreed-upon quality levels in the form of `Service-Level`

`Agreement (SLA)`s. Also, API consumers often expect the API provider to support them with integration activities.

(3) ***"The API provider has to treat the API as a product with a lifecyle"*** **(Bondel et al., 2022).**

An API makes functionality or data accessible to API consumers. The backend systems providing the functionality or data evolve and change continuously. On the other hand, API consumers evolve their clients and expect the API to change accordingly. Hence, the API between the backend systems and clients must change based on consumer wishes or technology trends. As a result, the API provider needs to actively manage changes to the API along a lifecycle in coordination with the backend provider and the API consumer.

(4) ***"The collaboration between the API provider team and all other stakeholders is challenging"*** **(Bondel et al., 2022).**

Collaboration between the API management team and internal stakeholders, e.g., the backend teams, primarily focuses on quality, defect, and incident management across teams, business units, or company boundaries. The interviewees repetitively stressed collaboration challenges, e.g., due to different priorities and timelines between these stakeholders. While some approaches exist to standardize the collaboration between these stakeholders, e.g., `API provider-wide ticketing management`, most collaboration relies on ad-hoc communication channels such as email.

In the future we plan on validating and extending this API management pattern catalog based on further insights into API initiatives.

APPENDIX A

Expert Interviews

Overall, 16 interviews with 15 interview partners informed the research approach. An overview of the interviews is presented in Tab. A.1 including information on the interviewees' role, the classification of the interviewees' employing organization, and the organizations' size.

#	Industry / Classification	Role	# Employees	Participants
1	Multi-banking startup	Backend Developer	11 - 50	IV1
2	Industrial manufacturing	Internal Consulting	>100.000	IV2
3	Automotive	Product Owner	>100.000	IV3, IV4
4	Software & IT service provider	Software Architect	1001 - 5000	IV5
5	IT service subsidiary	Portfolio Manager	1001 - 5000	IV6
6	Insurance subsidiary	Software Architect	51 - 250	IV7
7	Industrial manufacturing	Product Owner	>100.000	IV8
8	Industrial manufacturing	Software Architect	>100.000	IV9
9	Financial services	Software Developer	10.001 - 50.000	IV10
10	Software & IT service provider	Internal Consulting	5001 - 10.000	IV11
11	Software & IT service provider	Integration Architect	51 - 250	IV12
12	Automotive	Product Owner	>100.000	IV3, IV4
13	Software & IT service provider	Technical Lead, Product Owner	5001 - 10.000	IV13, IV14
14	Software & IT service provider	Software Architect	1001 - 5000	IV5
15	IT service subsidiary	Portfolio Manager	1001 - 5000	IV6
16	IT service subsidiary	Internal Consulting	1001 - 5000	IV15

Table A.1.: Overview of the interviews.

APPENDIX B

Case Base

The case base is derived from the interviews at the API portal level. We excluded cases representing strictly private API initiatives since they are out-of-scope of this pattern catalog. Furthermore, we added the two public API initiatives Stripe[1] and Twilio[2]. We reviewed these public API initiatives to validate previously identified patterns and add more details to the pattern descriptions, but we did not use them to mine new patterns. Tab. B.1 provides an overview of the case base.

[1]https://stripe.com/ (accessed 20.12.2023).
[2]https://www.twilio.com/de/ (accessed 20.12.2023).

Case Number	# Interview	Type of Initiative	Maturity	# of API Consumers	Used to
C1 (excluded)	1	Private	Development	<20	
C2	2	Partner	Pilot	<20	derive & validate patterns
C3	3, 12	Public & Partner	Production	>20	derive & validate patterns
C4	4, 14	Public	Production	>10000	derive & validate patterns
C5	4, 14	Partner	Production	>20	derive & validate patterns
C6	5, 15, 16	Group	Production	na	derive & validate patterns
C7	6	Group	Development	<20	derive & validate patterns
C8 (excluded)	7	Private	Development	>20	
C9	8	Public & Partner	Production	na	derive & validate patterns
C10	9	Partner	Production	na	derive & validate patterns
C11	9	Public & Partner	Production	>10000	derive & validate patterns
C12	10	Partner	Pilot	<20	derive & validate patterns
C13	11	Public & Partner	Production	<20	derive & validate patterns
C14	13	Public & Partner	Production	>10000	derive & validate patterns
C15 (excluded)	13	Private	Development	<20	
C16 (Stripe)		Public	Production	>10,000	validate patterns
C17 (Twilio)		Public	Production	>10,000	validate patterns

Table B.1.: Reviewed and extended overview of the case base (Bondel et al., 2022) used to mine API management patterns.

APPENDIX C

Related patterns and pattern languages/catalogs

We analyzed 15 software pattern collections and related them to our API management patterns presented in this technical report. The relations are described in the section "Other Related Patterns" of each pattern description. Fig. C.1 presents a visualization of all relations between the patterns.

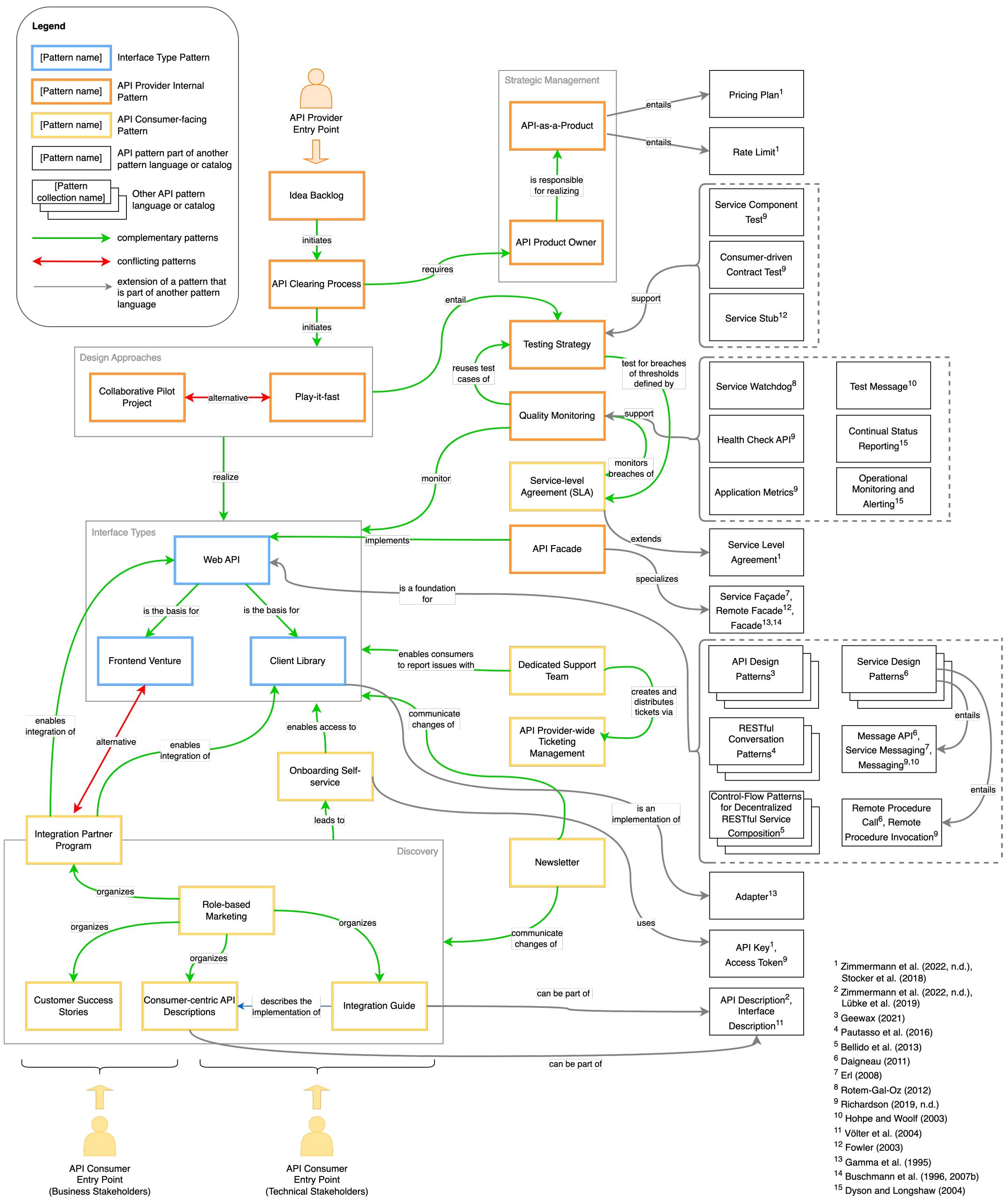
Legend
[Pattern name]
Interface Type Pattern
[Pattern name]
API Provider Internal Pattern
[Pattern name]
API Consumer-facing Pattern
[Pattern name]
API pattern part of another pattern language or catalog
[Pattern collection name]
Other API pattern language or catalog
complementary patterns
conflicting patterns
extension of a pattern that is part of another pattern language
API Provider Entry Point
Idea Backlog
initiates
API Clearing Process
initiates
requires
Strategic Management
API-as-a-Product
is responsible for realizing
API Product Owner
entails
entails
Pricing Plan[1]
Rate Limit[1]
Service Component Test[9]
Consumer-driven Contract Test[9]
Service Stub[12]
support
entail
Testing Strategy
reuses test cases of
test for breaches of thresholds defined by
Design Approaches
Collaborative Pilot Project
alternative
Play-it-fast
Quality Monitoring
support
monitors breaches of
Service Watchdog[8]
Health Check API[9]
Application Metrics[9]
Test Message[10]
Continual Status Reporting[15]
Operational Monitoring and Alerting[15]
Service-level Agreement (SLA)
monitor
realize
Interface Types
Web API
implements
API Facade
extends
Service Level Agreement[1]
specializes
Service Façade[7], Remote Facade[12], Facade[13,14]
is a foundation for
is the basis for
is the basis for
Frontend Venture
Client Library
enables consumers to report issues with
Dedicated Support Team
creates and distributes tickets via
API Provider-wide Ticketing Management
communicate changes of
API Design Patterns[3]
Service Design Patterns[6]
RESTful Conversation Patterns[4]
Message API[6], Service Messaging[7], Messaging[9,10]
entails
entails
Control-Flow Patterns for Decentralized RESTful Service Composition[5]
Remote Procedure Call[6], Remote Procedure Invocation[9]
enables integration of
alternative
enables integration of
enables access to
Onboarding Self-service
leads to
is an implementation of
Integration Partner Program
Discovery
Newsletter
Adapter[13]
organizes
Role-based Marketing
organizes
organizes
organizes
uses
communicate changes of
API Key[1], Access Token[9]
Customer Success Stories
Consumer-centric API Descriptions
describes the implementation of
Integration Guide
can be part of
API Description[2], Interface Description[11]
can be part of
API Consumer Entry Point (Business Stakeholders)
API Consumer Entry Point (Technical Stakeholders)
1 Zimmermann et al. (2022, n.d.), Stocker et al. (2018)
2 Zimmermann et al. (2022, n.d.), Lübke et al. (2019)
3 Geewax (2021)
4 Pautasso et al. (2016)
5 Bellido et al. (2013)
6 Daigneau (2011)
7 Erl (2008)
8 Rotem-Gal-Oz (2012)
9 Richardson (2019, n.d.)
10 Hohpe and Woolf (2003)
11 Völter et al. (2004)
12 Fowler (2003)
13 Gamma et al. (1995)
14 Buschmann et al. (1996, 2007b)
15 Dyson and Longshaw (2004)

Figure C.1.: Relation between patterns of this API management pattern catalog and patterns of other software pattern languages and catalogs.

List of Figures

2.1. Conceptual overview of software artifacts and stakeholders involved in API management (adapted from Bondel et al. (2022)). 9

3.1. Visualization Web API management pattern relations. 15

4.1. A `Web API` makes functionality and data accessible to API consumers via the HTTP protocol. 20
4.2. A `Client Library` wraps a Web API so that API consumers can access the API functionality and data using a specific programming language and framework. 26
4.3. A `Frontend Venture` makes data and functionality available to consumers lacking the capabilities to use a `Web API` or `Client Library`. 34

5.1. When applying the pattern `API-as-a-Product`, an API provider has to take technical, business, legal, and marketing aspects into consideration for each Web API. 40
5.2. An `API Product Owner` connects knowledge about the API provider organization's internal processes with knowledge of API consumers' needs. 49
5.3. The pattern `Collaborative Pilot Project` involves one or a selected number of API consumers in all steps of the API design. This includes the API consumer reviewing and providing feedback on the API's specification and the prototypical implementation before publication to all consumers. 54
5.4. If an API provider applies the `Play-it-fast Approach` pattern to design a new API, API consumers are not involved in the API specification and implementation. 59
5.5. The API provider collects and aggregates consumers' ideas for new APIs and changes to existing APIs in an `Idea Backlog`. 64
5.6. The API provider enforces a centrally defined `Testing Strategy` for new APIs or changes to existing APIs. 69
5.7. All new APIs or major changes need to pass the *API Clearing Process*. 74

5.8. The API Facade presents a unified interface to a set of backends (Gamma et al., 1994). 79
5.9. According to the pattern API Quality Monitoring, the provider continuously monitors the non-functional properties of an API in production. 85

6.1. According to the pattern Role-based Marketing, the API provider presents information tailored to different roles on an API portal landing page. 93
6.2. An API provider presents Customer Success Stories on the API developer portal. 98
6.3. The API provider creates a Newsletter to keep consumers up-to-date on changes.103
6.4. A Consumer-centric API Description lists the major use cases or user stories of the APIs on the developer portal. 108
6.5. The Integration Guide provides step-by-step instructions to implement a specific functionality. 113
6.6. If the API provider offers an *Onboarding Self-service*, the consumer does not have to interact with the provider before starting to use an API. 120
6.7. The provider presents a list of integration partners on the API developer portal according to the pattern Integration Partner Program. 128
6.8. The API consumers enter tickest into the API provider-wide ticketing management, which the system routes to the responsible teams within the API provider organization. 134
6.9. The Dedicated Support Team forms the interface between the API consumer and provider. The support team solves low- or medium-complexity tickets and forwards more complex requests to the respective experts in the provider organization. 139
6.10. An Service-Level Agreement (SLA) specifies the quality of services for an API.146

C.1. Relation between patterns of this API management pattern catalog and patterns of other software pattern languages and catalogs. 166

List of Tables

1.1. Overview of pattern languages and catalogs reviewed and related to patterns in this API management pattern catalog. 6

3.1. Summaries of the Interface Type Patterns. 16
3.2. Summaries of the API Provider Internal Patterns. 17
3.3. Summaries of the API Consumer-facing Patterns. 18

A.1. Overview of the interviews. 162

B.1. Reviewed and extended overview of the case base (Bondel et al., 2022) used to mine API management patterns. 164

Bibliography

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: Towns, buildings, construction.* (1st ed.). Oxford University Press.

Bellido, J., Alarcón, R., & Pautasso, C. (2013). Control-flow patterns for decentralized restful service composition. *ACM Trans. Web*, *8*(1). https://doi.org/10.1145/2535911

Bermbach, D., & Wittern, E. (2016). Benchmarking web api quality. In A. Bozzon, P. Cudre-Maroux, & C. Pautasso (Eds.), *Web engineering* (pp. 188–206). Springer International Publishing. https://doi.org/10.1007/978-3-319-38791-8_11

Bondel, G., Landgraf, A., & Matthes, F. (2022). Api management patterns for public, partner, and group web api initiatives with a focus on collaboration. *26th European Conference on Pattern Languages of Programs*. https://doi.org/10.1145/3489449.3490012

Bondel, G., Nägele, S., Koch, F., & Matthes, F. (2020). Barriers for the advancement of an api economy in the german automotive industry and potential measures to overcome these barriers. *Proceedings of the 22nd International Conference on Enterprise Information Systems*, 727–734. https://doi.org/10.5220/0009353407270734

Brandolini, A. (2013). Introducing event storming [accessed 20.12.2023]. http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html

Buckl, S., Ernst, A. M., Lankes, J., & Matthes, F. (2008). *Enterprise architecture management pattern catalog version 1.0* (tech. rep. No. 1) [Technical Report TB 0801]. Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München. Boltzmannstraße 3, 85748 Garching b. Münnchen, Germany.

Buckl, S., Matthes, F., Schneider, A. W., & Schweda, C. M. (2013). Pattern-based design research – an iterative research method balancing rigor and relevance. In J. vom Brocke, R. Hekkala, S. Ram, & M. Rossi (Eds.), *Design science at the intersection of physical and virtual design* (pp. 73–87). Springer Berlin Heidelberg.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007a). *Pattern-oriented software architecture: A pattern language for distributed computing* (1st ed., Vol. 4). John Wiley & Sons.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007b). *Pattern oriented software architecture: On patterns and pattern languages* (1st ed., Vol. 5). John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns* (1st ed., Vol. 1). John Wiley & Sons.

Christensen, C., Hall, T., Dillon, K., & Duncan, D. S. (2016). *Competing against luck: The story of innovation and customer choice* (1st ed.). Harper Business.

Cockburn, A. (2000). *Writing effective use cases* (1st ed.). Addison-Wesley Professional.

Cohn, M. (2004). *User stories applied: For agile software development* (1st ed.). Addison-Wesley Professional.

Cooper, A. (2004). *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity* (Vol. 2). Sams Indianapolis.

Coplien, J. O. (1996). *Software patterns*. SIGS Books & Multimedia.

Daigneau, R. (2011). *Service design patterns: Fundamental design solutions for soap/wsdl and restful web services: Fundamental design solutions for soap/wsdl and restful web services*. Addison Wesley.

Dal Bianco, V., Myllärniemi, V., Komssi, M., & Raatikainen, M. (2014). The role of platform boundary resources in software ecosystems: A case study. *2014 IEEE/IFIP Conference on Software Architecture*, 11–20. https://doi.org/10.1109/WICSA.2014.41

De, B. (2017). *Api management: An architect's guide to developing and managing apis for your organization* (1st ed.). Apress.

de Reuver, M., Sørensen, C., & Basole, R. C. (2018). The Digital Platform: A Research Agenda. *Journal of Information Technology*, *33*(2), 124–135. https://doi.org/10.1057/s41265-016-0033-3

Dyson, P., & Longshaw, A. (2004). *Architecting enterprise solutions - patterns for high-capability internet-based systems* (Vol. 1). John Wiley & Sons.

Eaton, B., Elaluf-Calderwood, S., Sørensen, C., & Yoo, Y. (2015). Distributed Tuning of Boundary Resources: The Case of Apple's iOS Service System. *MIS Quarterly*, *39*(1), 217–243. https://doi.org/10.25300/MISQ/2015/39.1.10

Erl, T. (2008). *Soa design patterns* (1st ed.). Pearson.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation).

Fowler, M. (2003). *Patterns of enterprise application architecture* (1st ed.). Addison Wesley.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* Prentice Hall.

Geewax, J. (2021). *Api design patterns* (1st ed.). Manning Publications.

Ghazawneh, A., & Henfridsson, O. (2010). Governing third-party development through platform boundary resources [http://aisel.aisnet.org/icis2010_submissions/48]. *ICIS 2010 Proceedings.*, 1–18.

Ghazawneh, A., & Henfridsson, O. (2013). Balancing platform control and external contribution in third-party development: The boundary resources model. *Information Systems Journal*, *23*. https://doi.org/10.1111/j.1365-2575.2012.00406.x

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., & Lafon, Y. (2007). *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* (W3C Recommendation). World Wide Web Consortium (W3C). World Wide Web Consortium (W3C). https://www.w3.org/TR/soap12-part1/

Henfridsson, O., Mathiassen, L., & Svahn, F. (2014). Managing technological change in the digital age: The role of architectural frames. *Journal of Information Technology*, *29*(1), 27–43.

Hevner, A. R. (2007). A three cycle view of design science research [https://aisel.aisnet.org/sjis/vol19/iss2/4]. *Scandinavian journal of information systems*, *19*(2), 4.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, 75–105.

Higginbotham, J. (2016). Sdk patterns for accelerating api integration [accessed 20.12.2023]. https://dzone.com/articles/sdk-patterns-for-accelerating-api-integration

Hofer, S., & Schwentner, H. (2021). *Domain storytelling: A collaborative, visual, and agile way to build domain-driven software* (1st ed.). Pearson International.

Hohpe, G. (n.d.). Enterprise integration patterns [accessed 20.12.2023]. %5Curl%7Bhttps://www.enterpriseintegrationpatterns.com/index.html%7D

Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions* (1st ed.). Addison Wesley.

IBM Cloud Education. (2021). Sdk vs. api: What's the difference? [accessed 20.12.2023]. https://www.ibm.com/cloud/blog/sdk-vs-api

Islind, A. S., Lindroth, T., Snis, U. L., & Sørensen, C. (2016). Co-creation and Fine-Tuning of Boundary Resources in Small-Scale Platformization. In U. Lundh Snis (Ed.), *Nordic Contributions in IS Research* (pp. 149–162). Springer International Publishing. https://doi.org/10.1007/978-3-319-43597-8_11

ITIL 4. (2019). *ITIL Foundation: ITIL 4 Edition* (tech. rep. No. 4). AXELOS Global Best Practices. Norwich, UK, TSO.

Jacobson, D., Brail, G., & Woods, D. (2012). *Apis: A strategy guide*. O'Reilly.

Karhu, K., Gustafsson, R., & Lyytinen, K. (2018). Exploiting and Defending Open Digital Platforms with Boundary Resources: Android's Five Platform Forks. *Information Systems Research*, *29*(2), 479–497. https://doi.org/10.1287/isre.2018.0786

Kendrick, T. (2015). *Identifying and Managing Project Risk: Essential Tools for Failure-Proofing Your Project* (3rd ed.). AMACOM.

Khosroshahi, P. A., Hauder, M., Schneider, A. W., & Matthes, F. (2015). *Enterprise architecture management pattern catalog version 2.0* (tech. rep. No. 1) [Technical Report]. Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München. Boltzmannstraße 3, 85748 Garching b. München, Germany.

Kircher, M., & Jain, P. (2004). *Pattern-oriented software architecture: Patterns for resource management* (1st ed., Vol. 3). John Wiley & Sons.

Landgraf, A. (2021). *Identification of api management patterns from an api provider perspective* (Master's thesis). Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München. Boltzmannstraße 3, 85748 Garching b. Münnchen, Germany.

Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., & Stocker, M. (2019). Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. *Proceedings* *of the 24th European Conference on Pattern Languages of Programs*. https://doi.org/10.1145/3361149.3361164

Maleshkova, M., Pedrinaci, C., & Domingue, J. (2010). Investigating web apis on the world wide web. *2010 Eighth IEEE European Conference on Web Services*, 107–114. https://doi.org/10.1109/ECOWS.2010.9

Medjaoui, M., Wilde, E., Mitra, R., & Amundsen, M. (2018). *Continuous API Management: Making the Right Decisions in an Evolving Landscape*. O'Reilly UK Ltd.

Meszaros, G., & Doble, J. (1997). A pattern language for pattern writing. *Proceedings of International Conference on Pattern languages of program design (1997)*, *131*, 164.

MuleSoft. (2021). *2021 connectivity benchmark report* (tech. rep.). MuleSoft. US, CA.

Newman, S. (n.d.). Sam newman & associates [accessed 20.12.2023]. https://samnewman.io/index.html

Newman, S. (2019). *Monolith to microservices - evolutionary patterns to transform your monolith* (1st ed.). O'Reilly Media.

OMG UML. (2017). *Omg® unified modeling language® (omg uml®) - version 2.5.1* (tech. rep.). Object Management Group.

Osterwalder, A., Pigneur, Y., Bernarda, G., & Smith, A. (2014). *Value proposition design: How to create products and services customers want* (1st ed.). John Wiley & Sons.

Pautasso, C., Ivanchikj, A., & Schreier, S. (2016). A pattern language for RESTful conversations. *Proceedings of the 21st European Conference on Pattern Languages of Programs*, 1–22. https://doi.org/10.1145/3011784.3011788

Pruitt, J., & Adlin, T. (2005). *The persona lifecycle: Keeping people in mind throughout product design* (1st ed.). Morgan Kaufmann Publishers Inc.

Red Hat. (2020). Was ist ein sdk? [accessed 20.12.2023]. https://www.redhat.com/de/topics/cloud-native-apps/what-is-SDK

Richardson, C. (n.d.). Microservices architecture: A pattern language for microservices [accessed: 20.12.2023]. %5Curl%7Bhttps://microservices.io/patterns/index.html%7D

Richardson, C. (2019). *Microservices patterns* (1st ed.). Manning Publications.

Rotem-Gal-Oz, A. (2012). *SOA Patterns* (1st ed.). Manning Publications.

Santoro, M., Vaccari, L., Mavridis, D., Smith, R., Posada, M., & Gattwinkel, D. (2019). *Web application programming interfaces (apis): General purpose standards, terms and european commission initiatives* (tech. rep. JRC118082). Publications Office of the European Union. Luxembourg. https://doi.org/10.2760/675

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-oriented software architecture: Patterns for concurrent and networked objects* (1st ed., Vol. 2). John Wiley & Sons.

Spichale, K. (2017). *Api-design - praxishandbuch für java- und webservice-entwickler* (1st ed.). dpunkt.verlag.

Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., & Pautasso, C. (2018). Interface quality patterns: Communicating and improving the quality of microservices apis. *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. https://doi.org/10.1145/3282308.3282319

Twilio. (2022). *Twilio build program guide for consulting partners - 2022 program year* (tech. rep.) [online resource https://twilio-cms-prod.s3.amazonaws.com/documents/BuildProgram-May2022Update-ConsultingGuide-r2_1.pdf]. Twilio.

Tynan, A. C., & Drayton, J. (1987). Market segmentation. *Journal of marketing management*, *2*(3), 301–335. https://doi.org/10.1080/0267257X.1987.9964020

Uebernickel, F., & Brenner, W. (2016). Design thinking. In C. P. Hoffmann, S. Lennerts, C. Schmitz, W. Stölzle, & F. Uebernickel (Eds.), *Business innovation: Das st. galler modell* (pp. 243–265). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-07167-7_15

Uludağ, Ö., Harders, N.-M., & Matthes, F. (2019). Documenting recurring concerns and patterns in large-scale agile development. *Proceedings of the 24th European Conference on Pattern Languages of Programs*. https://doi.org/10.1145/3361149.3361176

Vaccari, L., Posada, M., Body, M., Gattwinkel, D., Mavridis, D., Smith, R., Santoro, M., Nativi, S., Medjaoui, M., Reusa, I., Switzer, S., & Friis-Christensen, A. (2020). *Application programming interfaces in governments: Why, what and how* (tech. rep. JRC120429). Publications Office of the European Union. Luxembourg. https://doi.org/10.2760/58129

Völter, M., Kircher, M., & Zdun, U. (2004). *Remoting patterns: Foundations of enterprise, internet and realtime distributed object middleware* (1st ed.). John Wiley Sons Ltd.

Walker, G. (2001). *IT Problem Management* (1. Edition). Prentice Hall.

Wiesche, M., Jurisch, M. C., Yetton, P. W., & Krcmar, H. (2017). Grounded theory methodology in information systems research. *MIS Q.*, *41*(3), 685–701. https://doi.org/10.25300/MISQ/2017/41.3.02

Yoo, Y., Henfridsson, O., & Lyytinen, K. (2010). Research Commentary —The New Organizing Logic of Digital Innovation: An Agenda for Information Systems Research. *Information Systems Research*, *21*(4), 724–735. https://doi.org/10.1287/isre.1100.0322

Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., & Lübke, D. (2018). *Guiding architectural decision making on quality aspects in microservice apis*. Springer International Publishing. https://doi.org/10.1007/978-3-030-03596-9_5

Zimmermann, O., Lübke, D., Zdun, U., Pautasso, C., & Stocker, M. (2020). Interface responsibility patterns: Processing resources and operation responsibilities. *Proceedings of the European Conference on Pattern Languages of Programs 2020*. https://doi.org/10.1145/3424771.3424822

Zimmermann, O., Pautasso, C., Lübke, D., Zdun, U., & Stocker, M. (2020). Data-oriented interface responsibility patterns: Types of information holder resources. *Proceedings of the European Conference on Pattern Languages of Programs 2020*. https://doi.org/10.1145/3424771.3424821

Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., & Zdun, U. (2019). Introduction to microservice api patterns (map). *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. https://doi.org/10.4230/OASICS.MICROSERVICES.2017-2019.4

Zimmermann, O., Stocker, M., Lübke, D., & Zdun, U. (2017). Interface representation patterns: Crafting and consuming message-based remote apis. *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. https://doi.org/10.1145/3147704.3147734

Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., & Pautasso, C. (n.d.). Patterns for api design - simplifying integration with loosely coupled message exchanges [accessed: 14.04.2023].

Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., & Pautasso, C. (2022). *Patterns for api design: Simplifying integration with loosely coupled message exchanges* (1st ed.). Addison-Wesley Professional.