Patterns on Deriving APIs and their Endpoints from Domain Models

Apitchaka Singjai, Uwe Zdun
University of Vienna, Software Architecture Research Group, Vienna, Austria
firstname.lastname@univie.ac.at

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland (OST), Rapperswil, Switzerland
olaf.zimmermann@ost.ch

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland
cesare.pautasso@usi.ch

ABSTRACT

Domain-Driven Design (DDD) places the domain model at the center of all software development practices. Remote API design is crucial for developing distributed systems including, for example, microservice-based systems. While software practitioners realize APIs based on DDD models, clear guidance on how to derive APIs and API endpoints from domain model elements is still missing. Based on prior in-depth studies of practitioner sources on this and related topics, we have mined patterns to address these design problems. In particular, we present the DOMAIN MODEL FACADE AS API pattern which describes how to derive an API from a Domain Model. To explain further how derive API endpoints constituting the API from Domain Model elements, we present the AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS patterns. In addition, we relate these patterns to the previously published patterns API DESCRIPTION and API CONTRACT, both explaining how to describe APIs formally.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*;

KEYWORDS

Domain Driven Design, Microservice API, Design Patterns

ACM Reference Format:
Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann, and Cesare Pautasso. 2021. Patterns on Deriving APIs and their Endpoints from Domain Models. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP'21), July, 2021, Irsee, Germany*. ACM, New York, NY, USA, Article 4, 15 pages. https://doi.org/10.1145/3282308.3282324

1 INTRODUCTION

Domain-Driven Design (DDD) [8, 35] is a design approach that places the (business) domain at the center of software designing and architecting. Design in DDD leads to the rigorous modeling of a DOMAIN MODEL [10] and using it to build a UBIQUITOUS LANGUAGE that enables software development teams to use domain terminology throughout the software systems we build. Evans [8] classifies domain objects into types such as ENTITIES, VALUE OBJECTS, and SERVICES, which are then used to identify larger structures such as AGGREGATES. At the next higher abstraction level, DDD introduces the notion of *Strategic Design* [8, 35] which explains how to structure large domains into a number of BOUNDED CONTEXTS and their relations – modeled in so-called CONTEXT MAPS.

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [40]. They are often identified based on DDD models. They typically communicate via *APIs* in a loosely coupled fashion. Those remote APIs can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or remote procedure call technologies such as gRPC. A critical aspect in designing a microservice architecture is API design which includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to represent API messages, and so on [45].

APIs are often used as the externally visible interfaces of systems modeled with DDD. Thus, the question arises how to derive APIs from DDD models. In our prior works, we investigated the interrelation of microservice API design and DDD [29], and DDD violations in the context of coupling smells [28]. Those smells and violations are especially problematic when used in distributed setting. Lastly, we have modeled the DDD to API mappings of so far 14 system descriptions and open source systems by practitioners as UML models. Based on the data sets created in those studies, we have mined the patterns presented in this work.

Our main contributions are: Based on the previously published patterns API DESCRIPTION and API CONTRACT, we explain how they can help in answering the design question *how to formally describe the API*. Next, we present a pattern on *how to derive APIs from a DOMAIN MODEL and its elements*. We identified the DOMAIN MODEL FACADE AS API pattern which derives an API as a FACADE [11] to an identified subset of DOMAIN MODEL elements well-suited to be exposed to the API[1] We describe it along with 4 pattern variants and two regularly occurring alternative practices. Next, we present patterns on *how to derive API endpoints from domain model elements*. For this, we identified the AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS patterns. They all describe how to derive API endpoints from specific kinds of domain model elements. We present them

EuroPLoP '21, July, 2021, Irsee, Germany

ACM ISBN 978-1-4503-6387-7/18/07…$15.00
https://doi.org/10.1145/3282308.3282324

[1] Please note that we use the term "exposed to API" to indicate that one or more domain model elements are formally mapped to corresponding API elements, e.g. in a model or by implementation.

together with 2 regularly occurring alternative practices (ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS). Please note that in this work API endpoints cover all kinds of APIs realized with different kinds of technologies including RESTful HTTP, gRPC, SOAP, and messaging endpoints.

The target audience of this work are software/API developers and architects who are interested in the relations of DDD and APIs, as well as software engineering researchers studying those concepts.

This article is structured as follows: First, we discuss the related work in Section 2. After that we explain our research method in Section 3. Next, we motivate the need for new patterns by discussing common API design smells in Section 4. Section 5 then discusses the patterns on deriving APIs from DDD models. Finally, in Section 6 we draw conclusions.

2 RELATED WORK

This section outlines and compares to relevant related works. There are a substantial number of patterns and pattern languages in closely related areas. Firstly, core works on DDD such as those by Evans [8] and Vernon [35] describe DDD practices as patterns. Also various distributed systems patterns exist, too. The closest are our Microservice API patterns [22, 46] which describe best practices on the design of microservices APIs. In addition, patterns for various style of distribution have been discussed such as Messaging Patterns [17], Remoting Patterns [36], Enterprise Application Architecture patterns [10], Cloud Adoption Patterns [4, 30], and Service Design Patterns [6], to name just a few. Many of these works, hint at how to derive APIs and distributed systems in general from domain models, but so far this is not the core focus of any of these works.

This work is based on two of our prior studies. Firstly, we studied how practitioners understand coupling smells [28] and generated a Grounded Theory (GT) [5, 13] explaining those coupling smells and their relations. In this study we found a number of evidences which indicated a link between coupling smells and related software engineering principles to DDD. In essence, issues in DDD models can lead to coupling smells, and vice versa. This is interesting in the API context, as those smells and issues tend to become worse when they occur in a distributed setting, as discussed in Section 4. In distributed setting, it also needs to be considered that smells resemble established distribution patterns such as DATA TRANSFER OBJECT (DTO) [10], which can lead, if not carefully analyzed, to detrimental refactorings.

Secondly, we conducted a Grounded Theory study based on the grey literature mainly focusing on the interrelation of microservice API design and DDD [29]. We derived six architectural design decisions (ADDs) with 27 design options and 27 design drivers.

In another previous work [38] related to the Microservice API patterns [22, 46] we identified ADDs in the area of microservice API quality from the in-depth study of 31 widely used APIs and 24 specifications, standards, and technologies. We reported six ADDs with 40 decision options and 47 drivers. Context Mapper [18] is a tool for model-driven engineering that focuses on modeling based on strategic and tactical DDD patterns. Various mappings to microservice technologies can be generated including OpenAPI/Swagger interface descriptions.

Taibi and Lenarduzzi [34] define a number of microservice bad smells. As discussed in Section 4, some of those are relating bad smells and APIs. In this sense, this work also confirms our observation that coupling smells are relevant in the context of our work and may increase in their intensity in a distributed setting.

Stylos and Myers [32] categorized and organized API design decisions by conducting empirical research (in particular, a multivocal literature review). They investigated the literature in API usability, whereas we mainly focus on the interrelation between API and DDD.

Li and Chou [20] propose three design patterns for RESTful Web services based on a case study of computer-supported telecommunications application services. Their work concentrates on REST APIs, with basic abstraction such as session, event subscription and relationships using REST composition. Our focus is broader, as we concentrate on all kinds of API concepts and technologies.

Ayas et al. [1] conducted a Grounded Theory study to investigate the decision making in microservice migrations. Their data collection is from interviewing 19 participants, and evaluated by 52 professionals. They realized decision making on technical dimension that reflects the organizational and operational levels.

Brogi et al. [3] present a systematic literature review on design principles, architectural smells, and refactorings for microservices based on analysis of 54 sources. The paper identified many design smells and their resolution. The smell resolution in the paper primarily is on the infrastructure level (e.g., ESB rightsizing is suggested) and therefore complementary to our work.

There are number of API design patterns for specific domains, for example, northbound API of Software-Defined Networking (SDN) [21, 39], Internet of Things (IoT) [33], and biological data [37]. While API design in general has been studied, the specific relation of API design to design practices and models commonly used (such as those in DDD) is yet understudied. This is gap in the state-of-the-art led us to write our patterns on deriving APIs and API endpoints from DDD domain model elements.

Our work aims to present more general design patterns, for the specific problem of designing the combination of API and DDD. API endpoints definition in our context are broader than the key abstractions of REST resource specification, as e.g. in Fielding's work [9]. In REST, a resource being served usually refers to one or more nouns, whereas an endpoint is the location where a service can be accessed. In a non-REST context, API endpoints have various kinds of meaning, such as in URL (Uniform Resource Locator), where it is almost synonymous to an endpoint. The Microservice API patterns [22, 31, 46] use the following definitions: An API ENDPOINT is the provider-side end of a communication channel and a specification of where the API endpoints are located so that APIs can be accessed by API CLIENTS. Each endpoint thus needs to have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), as well as in HTTP-based SOAP or RESTful HTTP. Each API ENDPOINT belongs to an API; one API can have different endpoints. Our work has the goal to help in deriving API and API endpoints from domain model elements, which could potentially ease software engineering tasks and decision making, for instance in contexts such as migration to/of microservice architectures or API design.

Table 1: Comparison to Related Work

Work/Reference	Core Topics					Methodology
	DDD	API	Microservices	Smells	Decisions	
Singjai, Simhandl, and Zdun [28]	Yes	No	No	Yes	No	GT/Grey Literature
Singjai, Zdun, and Zimmermann [29]	Yes	Yes	Yes	Yes	Yes	GT/Grey Literature
Taibi and Lenarduzzi [34]	No	Yes	Yes	Yes	No	Interviews
Stylos and Myers [32]	No	Yes	No	No	Yes	Multi-vocal Literature Review
Li and Chou [20]	No	Yes	No	No	Yes	Case Study
Ayas, Leitner, and Hebig [1]	No	Yes	Yes	No	Yes	GT/Interviews
Brogi, Neri, Soldani, and Zimmermann [3]	No	Yes	Yes	Yes	No	Systematic Literature Review
Our work	Yes	Yes	Yes	Yes	No	Pattern Mining based on GT/Grey Literature Study

Table 1 summarizes the main related works. We compare the core topics of the works to our work's core topics, i.e. if they address DDD, APIs, Microservices, Smells, and Design Decision, as well as the used methodologies.

3 RESEARCH PROCESS AND METHODS

This section explains how we have mined patterns on deriving APIs and API endpoints from domain model elements based on data from grey literature studies. The *grey literature* [12, 25] is the main data source in our work. In software engineering, grey literature can be defined as "any material about software engineering that is not formally peer-reviewed nor formally published" [12]. We decided to study grey literature sources representing acknowledged practitioners' views on *the interrelation of distributed APIs and DDD*.

Figure 1 illustrates our research process and methods. The main knowledge sources used in this work have been gathered in two of our prior works [28, 29]. In the first of those, we studied 32 practitioner sources from the grey literature in depth [29] using the Grounded Theory (GT) research method [5, 13], a systematic research method for discovery of theory from data. We studied each knowledge source in depth, followed GT's coding process, as well as a constant comparison procedure to derive a model of architectural decisions on deriving APIs and API endpoints from domain model elements. Hentrich et al. provide details on how GT's coding process is mapped to pattern mining [15]. Riehle et al. [27] explain various such systematic pattern mining methods, and propose steps for discovering, codifying, evaluating, and validating the patterns during pattern mining. In GT-based pattern mining, those steps are embodied in the coding and constant comparison processes of GT. Parts of the ADDs from this prior study turned to problems, solutions, and forces of the pattern described this work.

Using the same research methods, we also studied coupling smells based on 48 practitioner sources [28]. This study revealed many relations of coupling smells and principle violations to DDD models, and vice versa. As those tend to become specifically problematic in distributed settings, many aspects in this study are core motivations of this work, as discussed in Section 4, and contribute to the key forces and consequences of our patterns. We also considered existing patterns and pattern languages to enhance and detail our patterns.

In addition, to those detailed studies, we have modeled the DDD to API mappings of so far 14 system descriptions and open source systems by practitioners as UML models. We have used those system models to confirm our patterns, and we also used them to describe known uses of the patterns.

Please note that we report various *Practices* for which we found evidence that they are widely used by practitioners. For some of those Practices we have found substantial evidence that they are widely used "Best Practices", and those are reported as *Patterns* in this paper. Others are reported just as *Practices* because either there is not yet enough evidence that they are indeed Patterns, or they are seen critical by a substantial fraction of our sources and/or only useful in very specific niches.

4 MOTIVATION

This section explains why coupling smells related DDD violations are problematic in distributed settings and their API as a motivation for this work. Section 4.1 provides the causes that lead to the effects in Section 4.2.

4.1 The Risk of an Anemic Domain Model and its Relation to APIs

One of the key *anti-patterns* discussed in the realm of DDD is ANEMIC DOMAIN MODEL[2]. This anti-pattern describes a DOMAIN MODEL that fails to combine data with logic processing it in its realization. These domain objects look at first glance like good domain abstractions, as they are named with nouns from the domain's problem space and are connected with detailed relationships. Digging deeper and inspecting the object's behavior, however, shows that the objects actually carry little to no rich domain behavior (or business logic). Instead there are a number of SERVICES which contain the domain logic and act upon the rather 'dumb' objects in the ANEMIC DOMAIN MODEL.

If an API is derived from such an ANEMIC DOMAIN MODEL, things are getting even worse. Often every ENTITY in the ANEMIC DOMAIN MODEL leads to deriving a corresponding service, e.g. a RESTful service. The bad design practices in the domain model lead to shallow API endpoints, where clients need to understand all the complexity in the backend. Transactional or data consistency boundaries between services are missing, often leading to situations where the client needs to take part in ensuring data consistency in the backend services or where consistency management in the backend is hard to realize well. Such designs lead to chatty APIs with bad performance and scalability. API's are becoming hard to understand, maintain, or evolve.

Our studies reveal that such problems often arise, when developers start with an API design and only then derive the DOMAIN MODEL. Consider you are designing a RESTful API first. Often simple, shallow ENTITIES with CRUD (Create, Read, Update, Delete)

[2]See e.g. https://martinfowler.com/bliki/AnemicDomainModel.html.

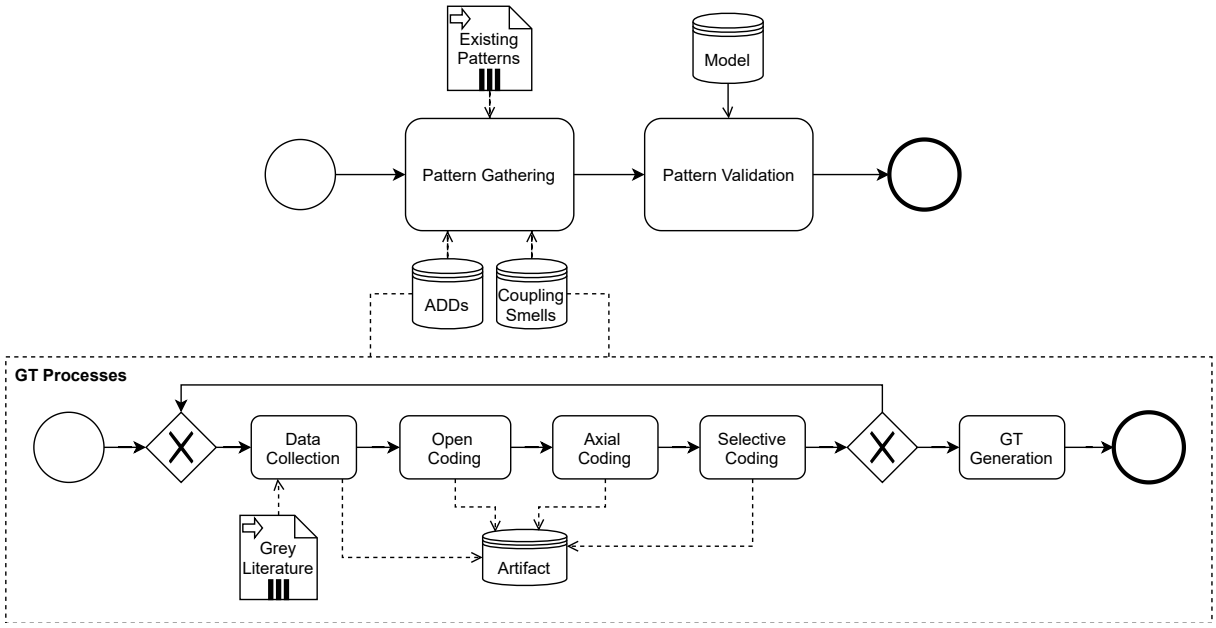

Figure 1: Research Overview

operations are designed initially because this seems natural and easy for a REST resource that refers to one or more nouns. When these are then modeled or implemented in a DOMAIN MODEL, it is likely that the initial result is an ANEMIC DOMAIN MODEL. If a substantial refactoring to rich API endpoints backed up by rich DOMAIN MODEL abstractions never happens, a shallow API on top of an ANEMIC DOMAIN MODEL is the consequence, with all kinds of negative consequences, such as the ones described above.

4.2 Relations to Coupling Smells

Our prior studies show that coupling issues in APIs and DDD models do not remain at the model level only. They transcend all the way down into the backend services' detailed design and code. For instance, we have found that *Coupling Code Smells* can cause and can be caused by ANEMIC DOMAIN MODEL related problems [28], and thus bad API designs as well. In our prior study on coupling smells [28], a number of practitioner sources relate coupling smells to issues in DOMAIN MODEL design. That is, ANEMIC DOMAIN MODELS and coupling smells often occur together in designs and might cause each other. This, in turn, has negative consequences for the API design and backend services realizing the API. Let us illustrate a few of those relations with their consequences:

- The *Data Class* bad smell describes a class that only offers data. This is actually the situation that leads to ANEMIC DOMAIN MODEL, and, if exposed to an API, this leads to shallow ENTITIES only with CRUD operations on them. As discussed, this leads to problems related to API complexity, data consistency, chatty APIs, performance and scalability issues, API understandability, API maintainability, and API evolvability.
- The *Feature Envy* bad smell describes a class or method that makes excessive use of a target class or its methods. This practice can lead to ANEMIC DOMAIN MODELS as classes do not combine data with logic processing it but rely on other classes excessively instead. This is especially problematic if the classes contributing to *Feature Envy* are distributed API endpoints, as excessive distributed calls lead to chatty APIs, performance and scalability issues, high API complexity, and interaction protocols that are hard to understand.
- The *Inappropriate Intimacy* bad smell describes a class using another class's implementation details. When this happens across API endpoint boundaries, it is even discussed specifically as a microservice bad smell by Taibi et al. [34], leading to similar issues as can be observed for *Feature Envy*.
- The *Message Chain* bad smell describes designs containing a long sequence of method calls. If this occurs in clients as calls to an API, this leads to many distributed calls, which is a symptom of hard to understand, complex APIs with bad performance (a so-called chatty API).
- The *Indecent Exposure* bad smell describes a class that exposes internal detail. If this class is exposed to the API, similar intimate call dependencies as in *Inappropriate Intimacy* or *Feature Envy* arise, with similar consequences.
- The *Middle Man* bad smell describes a class that only delegates to other classes. Again, in a distributed setting this can lead to many unnecessary distributed calls, which is a symptom of hard to understand, complex APIs with bad performance.

Our patterns introduced below help to avoid API designs that lead to or are caused by ANEMIC DOMAIN MODEL. Thus they help to spot and fix, or avoid in first place, the coupling smells and their consequences listed above. As a result, many of the forces and consequence of our patterns are directly related to the possible negative consequences of the ANEMIC DOMAIN MODEL and the coupling smells.

This discussion should show that API design requires careful DOMAIN MODEL design, but also intimate relations to the backend designs of the services realizing the API endpoints is needed. That is, the patterns can only be applied well, if related coupling smells and similar issues in the backends are resolved. Likewise, designs without prevalent coupling smells and similar design issues in the backends, are usually easier and better mappable to APIs.

5 API DERIVATION PATTERNS

Figure 2: Overview of the patterns for API descriptions and/or contracts

We first report on two previously mined patterns, API DESCRIPTION and API CONTRACT. We discuss them rather briefly, as API DESCRIPTION [22] has been published as part of the Microservice API Patterns [46] before, and API CONTRACT is just a variant of the INTERFACE DESCRIPTION pattern in the Remoting Patterns [36]. The patterns' relationships are illustrated in Figure 2.

Next, we present the DOMAIN MODEL FACADE AS API pattern that describes how to derive an API from a DOMAIN MODEL and its elements. The pattern explains in its pattern text a number of variants on how to achieve deriving API elements from a DOMAIN MODEL (namely *Domain Model to API Model Mapping*, *Interface Bounded Context*, *Shared Kernel Based Interface*[3], and *Implicit Designation of API Model Subset*). We also describe two alternative practices[4] (namely *Expose the Whole Domain Model 1:1 as the API* and *Expose Each Bounded Context as its Own API*). An API design based on DOMAIN MODEL FACADE AS API can or cannot use the API CONTRACT and/or API DESCRIPTION for API documentation. Figure 3 illustrates those pattern relations. The DOMAIN MODEL FACADE AS API should use API names and abstractions from the UBIQUITOUS LANGUAGE [8] formally specified by the DOMAIN MODEL. This makes it easy to trace from API elements to the corresponding domain model elements, and enable domain experts to understand the API. An API is a PUBLISHED LANGUAGE [8] in DDD terminology. If an API CONTRACT or API DESCRIPTION is used to specify the API design, the API CONTRACT or API DESCRIPTION is a formal specification of the PUBLISHED LANGUAGE.

The following patterns AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS describe how to derive API endpoints from domain model elements. We describe AGGREGATE ROOTS AS API ENDPOINTS in detail. For the very similar, but less often used DOMAIN SERVICES AS API ENDPOINTS and DOMAIN PROCESSES AS API ENDPOINTS patterns, we describe the differences to AGGREGATE ROOTS AS API ENDPOINTS briefly. Finally, in the pattern text of AGGREGATE ROOTS AS API ENDPOINTS we describe the two alternative practices ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS which only rarely deliver results on deriving an API from DOMAIN MODEL elements that balance the forces well. AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices) can use API CONTRACT or API DESCRIPTION to specify the mapped API design as well as the API endpoints formally. Figure 4 illustrates those pattern relations.

Figure 4 also shows the relation of the API endpoints patterns to DOMAIN MODEL FACADE AS API: An API is composed of a number of endpoints. Thus, the API endpoint patterns determine the domain model subset to be exposed in the API. Please note that there is a certain overlap between the DOMAIN MODEL FACADE AS API patterns and the API endpoints patterns, as some APIs are based on BOUNDED CONTEXTS, especially if dedicated interface BOUNDED CONTEXTS are designed with the purpose of creating APIs from them. But smaller, more limited, or only partly exposed BOUNDED CONTEXTS are also sometimes used as basis for identifying API endpoints.

5.1 Patterns: API Contract and API Description

An API DESCRIPTION [22, 46][5] contains the API CONTRACT that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between API provider and client. In addition to this syntactical interface description it contains quality management policies as well as semantic specifications and organizational information. The API CONTRACT part is essentially a special case or variant of the INTERFACE DESCRIPTION pattern [36] with the purpose of describing a remote API. For example, Swagger/Open API, WADL, or WSDL are INTERFACE DESCRIPTION languages used to describe API CONTRACTS. Please refer to the original pattern sources [22, 36, 46] for a detailed description of these patterns.

5.2 Pattern: Domain Model Facade as API

Alias. Remote Service Layer.

Context. In a software development project, you use DDD to design your DOMAIN MODEL and want to expose some parts of the software system you develop as an API.

Problem. How to derive an API from a DOMAIN MODEL?

Forces.

- *Avoiding Brittle Interfaces*: A naive solution is to map every element of the DOMAIN MODEL to the API. This can lead to brittle interfaces as any change in the DOMAIN MODEL leads to a change in the API. DOMAIN MODEL and backend implementation should be kept flexible, while APIs should usually – as far as possible – stay stable. Of course, as a downside, the more the DOMAIN MODEL and the API drift apart, the higher the effort for API design and the more difficult it might become to understand and keep track of the mapping between them. A good balance has to be found.
- *API Complexity*: Low complexity of the API is essential to make it comprehensible both for consumers of the API and API developers (see [2] for a definition of complexity as it is used here).

[3]Please note that a SHARED KERNEL can be defined as the shared interface that constitutes e.g. a solution internal APIs. For external or public APIs, usually rather interface BOUNDED CONTEXTS are used.

[4]We use the term "practice" if we found in our prior studies practices that are often applied by practitioners, but that should not be called a pattern or best practice. That is, they might work well in some design situations, but might in others be considered an anti-pattern.

[5]See https://microservice-api-patterns.org/patterns/foundation/APIDescription

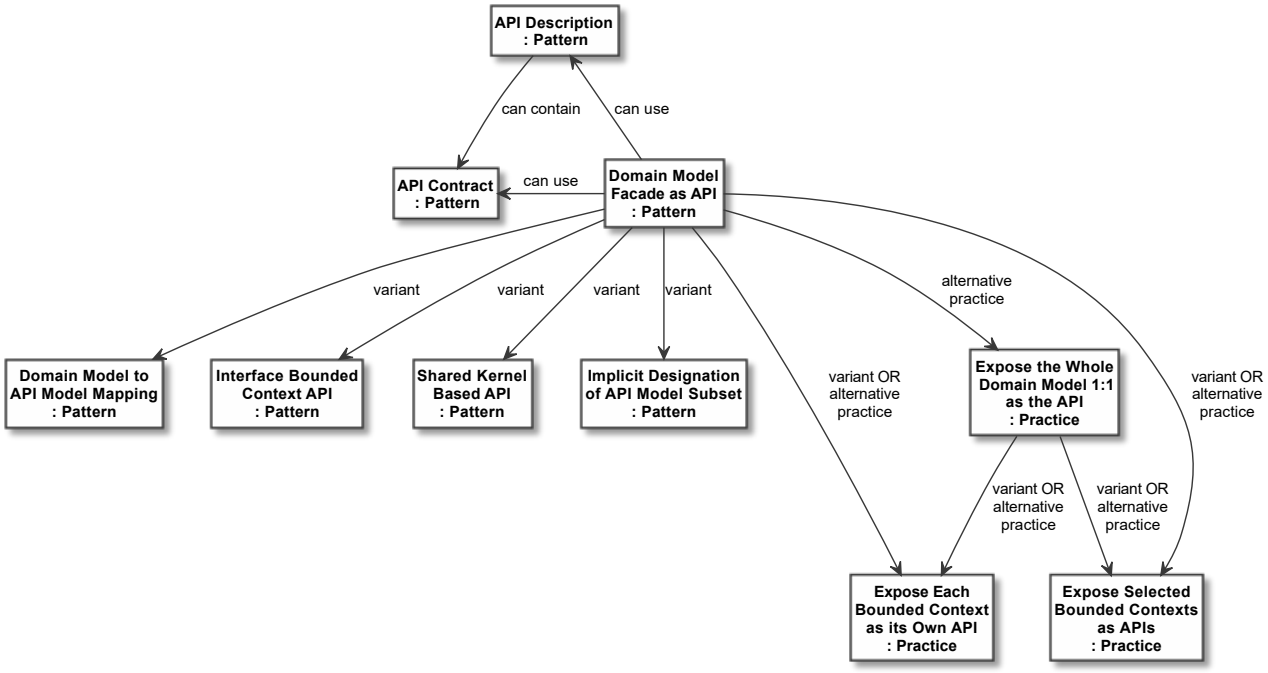

Figure 3: Overview of the patterns for how to derive an API from a DOMAIN MODEL

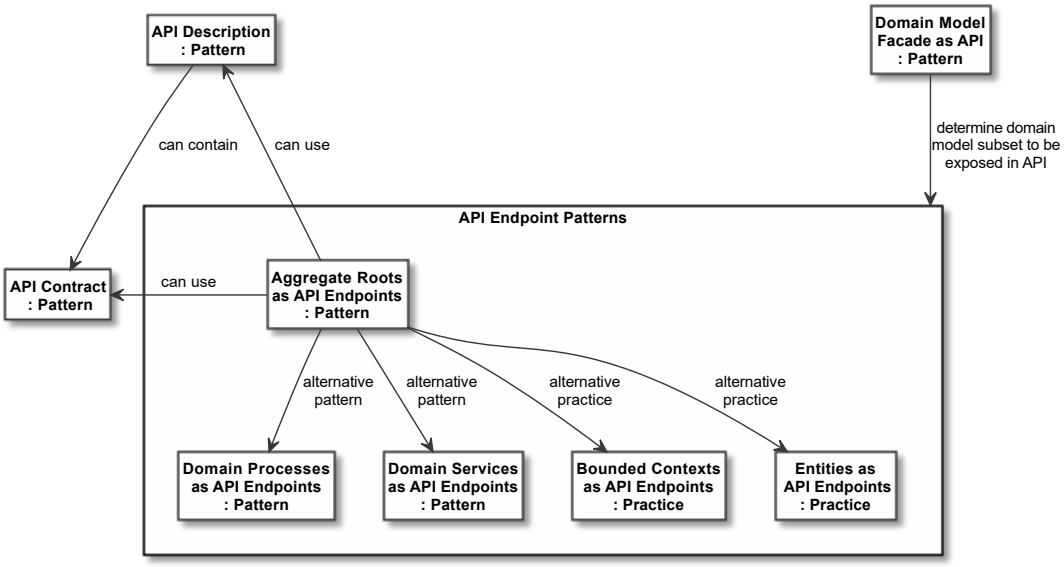

Figure 4: Overview of the patterns for how to derive API endpoints from domain model elements

Exposing all elements of a DOMAIN MODEL would lead to unnecessary complexity of the API, if a smaller or more simplified abstraction is possible. However, the more the DOMAIN MODEL and the API drift apart, the more complex the mapping logic becomes, also possibly leading to complexity. A middle ground where both API complexity and mapping complexity are kept in check have to be found. Following the POINT principles can help here[6]

- *API Usability and Understandability*: Aiming for low complexity and high stability means to make the API more usable to the consumer. As API consumers are often not experts for the backend

[6]See https://medium.com/olzzio/apis-should-get-to-the-point-c79113efa31c.

implementation, any unnecessary details should be abstracted away from the API. Also, those properties make an API understandable both for API consumers and API developers. Ideally, API consumers should not have to know any details about the underlying domain model. However, any such measures require additional design effort for the API, and it must be decided if they really pay off. A good API design should not come at the cost of a bad DOMAIN MODEL or backend design.

- *Modifiability*: Most software systems have to change over time. As pointed out, the pace of changes in DOMAIN MODELS and APIs can differ significantly, as APIs should rather stay stable, whereas DOMAIN MODELS must change as the requirements change. This can lead to tensions between the two abstractions. Also, some DOMAIN MODEL changes lead to required API changes. Such changes should rather be localized both in the DOMAIN MODEL and in the API. Finally, the API can change independently of the DOMAIN MODEL. For example, if the DOMAIN MODELS is gradually exposed to the API. Or, if technology changes induce API changes, the API can change without a change in the DOMAIN MODEL.
- *Design and Implementation Effort*: As designer and developer time is costly, usually only a limited amount of design and implementation effort can be invested in the initial realization of an API and its evolution. As mentioned above, many desired API features require additional design and implementation effort.
- *API Maintainability*: High complexity, low usability and understandability, and low modifiability can all lead to maintainability problems. Again, as a downside, those might be lead to higher design and implementation efforts throughout the API evolution. [24]
- *Security and Privacy*: Security and privacy can be an important consideration when deriving an API from a domain model, too. Interesting considerations to be made during API design can be, for instance, to avoid exposing confidential domain elements, enabling auditability, and supporting API monitoring or observability.

Solution. Introduce a dedicated API view on selected parts of the DOMAIN MODEL to establish a PUBLISHED LANGUAGE that exposes parts of the UBIQUITOUS LANGUAGE of the domain in a controlled, managed fashion. Mark the domain model elements exposed to the API clearly as API elements.

Solution Details. In DDD terminology an API is a PUBLISHED LANGUAGE offered by the API publisher's contexts to the API consumers' contexts (both of these kinds of contexts can be modeled as BOUNDED CONTEXTS). The parts of the DOMAIN MODEL selected to be exposed in the API are usually determined based on the coarse-grained parts required by API consumers. Here, especially a selection of domain model elements to be exposed as API endpoints is necessary. How to do the selection of individual API endpoints candidates well is explained in the patterns AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS. Occasionally, the ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS patterns may also deliver good results on deriving an API from domain model elements, as explained above. Based on the to-be exposed coarse-grained domain model elements, it is necessary to determine which of them should be exposed as API endpoint in which API. That is, not always all endpoints of an application are exposed in the same API. Just consider an application that needs an API for ordinary API consumers plus an administration API interface. Then, two APIs exposed by the application need to be derived from the set of API endpoints of one application.

The design process is usually iterative; this is emphasized strongly in the literature on object-oriented analysis and design [19] and in the Design Practice Repository [43] that collects general purpose activity descriptions and artifact templates. It can start by considering API endpoints candidates using patterns such as AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS. As explained above, this is intertwined with the design of the *Application Services* in a SERVICE LAYER. The design process can also start by considerations about the whole API, e.g. the design of one or more DOMAIN MODEL FACADE AS APIS that are needed by API consumers. Usually, multiple iterations of the design are needed to find a good compromise.

In addition to coarse grained domain model elements, also more fine-grained elements need to be exposed. For example, messages and message contents of the API can be derived from links, data types, and operations.

Not every part of the API must have a representation in the DOMAIN MODEL. For example, DATA TRANSFER OBJECTS [10] might be introduced covering multiple domain model elements or only parts of some of them. But in order to make the API understandable to domain experts, it is essential that names and abstractions in the API follow the terms defined in the UBIQUITOUS LANGUAGE which is formally specified by the DOMAIN MODEL. Thus, any deviation from the DOMAIN MODEL should have a good reason.

Example. To illustrate the pattern let us again consider the excerpt of a DDD model of a publication management system from Section 5.3.

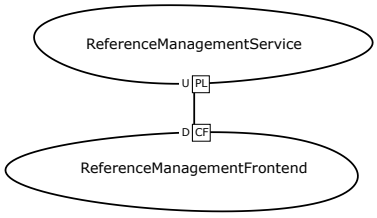

Figure 5: Publication Management Context Map (adapted from [41])

As shown in Figure 5, the *Reference Management Service* BOUNDED CONTEXT is a PUBLISHED LANGUAGE (indicated by *PL* in the CONTEXT MAP) for the *Reference Management Frontend* BOUNDED CONTEXT, which has just a CONFORMIST relation (indicated by *CF* in the CONTEXT MAP) to the Service BOUNDED CONTEXT. A CONFORMIST relation means that the downstream context (indicated by *D* in the CONTEXT MAP), which is dependent on the upstream context (indicated by *U* in the CONTEXT MAP), "eliminates the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team" [8]. That is, here it would not make much sense to expose

any domain model elements in the *Reference Management Frontend* BOUNDED CONTEXT as they would not contribute anything new to the API. In the context of the *Reference Management Service* BOUNDED CONTEXT it may also not make sense to see all elements of the DOMAIN MODEL as equally important for being exposed in the API. So instead of selecting the whole domain model as the scope making up the API, or creating APIs based on all the BOUNDED CONTEXTS, here a better choice is to select only the *Reference Management Service* BOUNDED CONTEXT as the scope for the API.

Next, we select the specific elements to be exposed to the API in this BOUNDED CONTEXT. In Figure 10 an API mapping is shown where the *Reference Management Service* BOUNDED CONTEXT is exposed as the *API* scope. This is denoted by the BOUNDED CONTEXT being exposed to the *Reference Management Service API*. This API offers an API endpoint which is exposed by the *Paper Archive Facade* AGGREGATE. The *Paper Item* and *Paper Item Key* domain model elements are both exposed as *API Data Types*. Via the *«exposed to API as»* relations, the subset of the API that is exposed to the API is clearly designated in this model.

Pattern Variants and Alternative Practices. There are a number of variants of DOMAIN MODEL FACADE AS API on how to mark elements clearly as API elements, as shown in Figure 6. The variants and alternative practices are used here to explain different ways how the pattern is applied in practice.

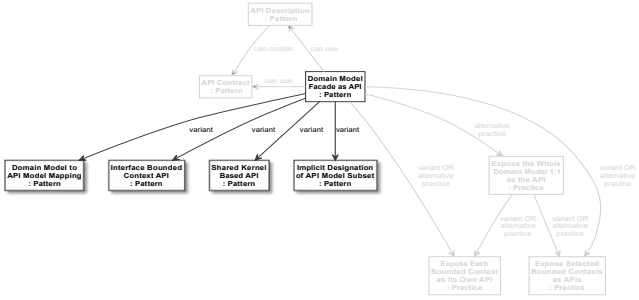

Figure 6: Domain Model Facade as API – Main Variants

- *Domain Model to API Model Mapping:* One option is to model the API explicitly alongside the domain model. Then the API model elements can be derived from domain model elements. Those can for instance be highlighted using dedicated stereotypes or relationship types.
- *Interface Bounded Context:* Another option is to design an explicit BOUNDED CONTEXT that contains the interface to be exposed in the API. This BOUNDED CONTEXT is designated as an Interface Bounded Context in the DOMAIN MODEL design.
- *Shared Kernel Based Interface:* A similar option is to design a SHARED KERNEL [8] for the interface between API consumer and publisher. The content of this SHARED KERNEL designates the domain model elements to be exposed in the API. This option assumes that the client scope is and can be modeled, too. For example, for public APIs this might be hard to do; but in more closed settings, e.g. where the teams developing the clients are known partners, this option is applicable. As a consequence, a SHARED KERNEL based interface would be used e.g. in a solution internal APIs. For external or public APIs, usually rather interface BOUNDED CONTEXTS are used.
- *Implicit Designation of API Model Subset to be Exposed in the Facade:* While it is advisable to mark domain model elements clearly as API elements, often less clear options than dedicated stereotypes or relationship types are chosen. For example, a variant of the pattern just uses the same or similar names for domain model element and API element. Such practices can lead confusions and inconsistencies in the mappings.

Two other *variants* of DOMAIN MODEL FACADE AS API, as shown in Figure 7, are to *Expose Each Bounded Context as its Own API* [29] or *Expose Selected Bounded Context as APIs* [29]. These two practices select the BOUNDED CONTEXT as API scope. They are a variant of DOMAIN MODEL FACADE AS API if the DOMAIN MODEL of each BOUNDED CONTEXT is exposed to the API following the guidances provided here in the DOMAIN MODEL FACADE AS API pattern.

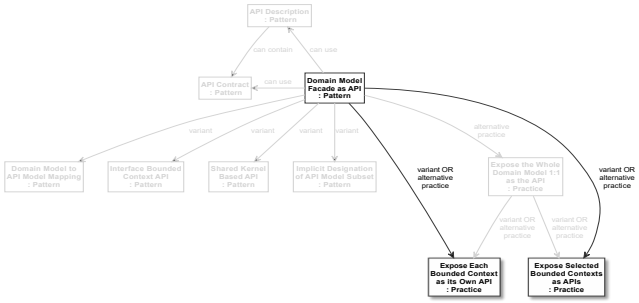

Figure 7: Domain Model Facade as API – Other Variants

There are two possible alternative solutions, as shown in Figure 8, sometimes discussed by practitioners, but which only rarely lead to acceptable results:

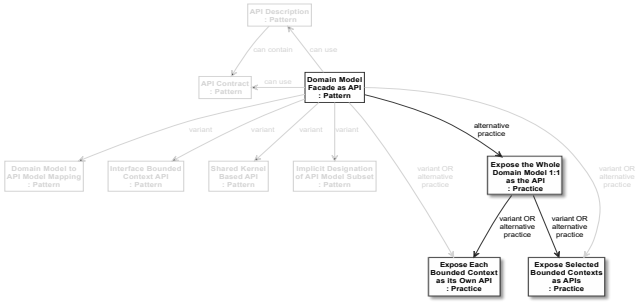

Figure 8: Domain Model Facade as API – Alternative Practices

- One alternative to applying this pattern is to *Expose the Whole Domain Model 1:1 as the API* [29]. At least in non-trivial domains, this solution is in many situations rather an anti-pattern. For example, it can lead to brittle interfaces, high API complexity, understandability issues, and API modifiability problems.

- The two practices *Expose Each Bounded Context as its Own API* or *Expose Selected Bounded Context as APIs* are merely alternative practices to the DOMAIN MODEL FACADE AS API pattern, if there API mapping is not following the DOMAIN MODEL FACADE AS API pattern. Especially, if their mapping is akin to *Expose the Whole Domain Model 1:1 as the API*, i.e. exposing the whole domain model of each of those BOUNDED CONTEXTS to the API, it is again rather an anti-pattern. That is, while those alternatives are then more fine-grained than *Expose the Whole Domain Model 1:1 as the API*, still all elements of those BOUNDED CONTEXTS are exposed, including many elements that do not have to be exposed. Thus those solutions tends to lead to similar problems as exposing the whole DOMAIN MODEL 1:1 as the API. For these reasons, *Expose Each Bounded Context as its Own API* or *Expose Selected Bounded Context as APIs* are shown in Figure 3 as *variant OR alternative practice* both for DOMAIN MODEL FACADE AS API and *Expose the Whole Domain Model 1:1 as the API*, depending on how the mapping of the DOMAIN MODELS of the API BOUNDED CONTEXTS is done.

Consequences.

+ *Stable Interfaces:* The pattern leads to more stable interfaces than solutions like exposing the whole DOMAIN MODEL or exposing all BOUNDED CONTEXTS as the API scope. The reason is that only selected interface elements are exposed, meaning that changes in other parts of the DOMAIN MODEL do not affect the API.
+ *API Complexity:* The pattern has a positive effect on API complexity as specific elements of the DOMAIN MODEL are selected and designed to be part of the API. Thus the API gets smaller and tends to contain abstractions and aggregated elements rather than every DOMAIN MODEL detail.
+ *API Usability and Understandability:* The pattern is beneficial to API usability and understandability because of the lowered complexity and increased stability it offers.
+ *API Modifiability:* Detailed selection of API-exposed domain elements is positive for API modifiability as it is easy to change and extend the selection.
+ *Avoiding API Maintainability Issues and Reducing Design and Implementation Effort:* As the pattern can lower complexity, improve usability and understandability, and improve modifiability it can help to avoid API Maintainability problems. This can lead to less design and implementation effort throughout the API evolution.
+ *Traceability:* If a systematic and formal mapping between domain model elements and API elements is used, the pattern enables traceability from API elements to contributing domain model elements.
+ *Security and Privacy:* This pattern creates a clear mapping between DOMAIN MODEL and API, making it easier to trace which parts of the domain model (at a coarser-grained level) are exposed via the API. This can help to fulfill auditability and/or monitoring requirements e.g. for possibly confidential parts of the domain model.
– *Design and Implementation Effort:* Compared to naively exposing the whole DOMAIN MODEL or exposing all BOUNDED CONTEXTS, the pattern requires initially a higher design and implementation effort.
– *API Usability, Understandability, Modifiability, and Maintainability:* Compared to the INTERFACE BOUNDED CONTEXT or SHARED KERNEL BASED INTERFACE variants, described above, the detailed selection of exposed elements has a negative effect on API usability and understandability, as well as API Maintainability, as the mapping might require both for the consumers and maintainers more effort for comprehending the API. This can be negative for API modifiability, too.
– *Clients Might Have to Manage Crossing Model Boundaries:* Another possible downside of this solution is that clients might have to manage crossing model boundaries, i.e., the boundaries between the BOUNDED CONTEXTS, but only if domain model elements from different contexts are exposed.
– *Traceability:* If no systematic and formal mapping between domain model elements and API elements is used, traceability can be lost, meaning that inconsistencies and confusion can arise. This can mean that many of the benefits of the pattern cannot be achieved.

Related Patterns. The DOMAIN MODEL FACADE AS API pattern describes how to establish one or more APIs based on the domain model elements to be exposed to an API through *Application Services* [35]. Together the *Application Services* form a SERVICE LAYER [10]. The SERVICE LAYER defines an application's boundary as a layer of (micro)services that implement the API. An API defines the client-visible interfaces of a subset of those services exposed to API consumers. For example, an application might define five *Application Services* in its SERVICE LAYER, each with its own endpoint. It might further define two APIs, one for ordinary API consumers and one an administration API; the first API exposes three of the *Application Service* endpoints, the second one the two other endpoints. The AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS patterns, explained later on ,as well as their alternative practices, are ways how find candidates for the mapping of coarse-grained domain model elements to API endpoints.

A REMOTE FACADE "provides a coarse-grained facade on fine-grained objects to improve efficiency over a network." [10]. Typically a DOMAIN MODEL FACADE AS API is special form of REMOTE FACADE which does not directly expose objects, but rather *Application Services* in the SERVICE LAYER [10] which then access objects and other implementation structures in their own service implementations or in backends. The relations to these patterns, is the reason for the alias *Remote Service Layer* of the DOMAIN MODEL FACADE AS API pattern.

Please note that DOMAIN EVENTS [8] and event-driven architecture related patterns such as EVENT SOURCING [26] or CQRS [26] are often important in API derivation. That is, if the backends use events and event-driven architectures, some of those or abstractions of those events can be exposed in the API. However, this is usually in first place a consideration of more detailed API design than the patterns covered in this paper (see [29] for more detailed architecture design decisions on this).

The API GATEWAY pattern [26] is often the place where the API is technically offered. Sometimes different APIs are offered via different gateway, e.g. as in the BACKENDS FOR FRONTENDS pattern [26].

In such cases, multiple APIs need to be derived from the same domain model. That is, the DOMAIN MODEL FACADE AS API is often useed to design the APIs exposed on API GATEWAY or BACKENDS FOR FRONTENDS.

Known Uses.

- The publication management system [41] used as an example above uses the pattern to select specific API elements that are described via the API DESCRIPTION language MDSL. MDSL can be used to generate the API CONTRACT in respective technologies such as Swagger/Open API, and/or generate models and code.
- In an online shop system model [44] that is also specified with MDSL, an AGGREGATE is selected as the scope for an API, and three endpoints are defined as a DOMAIN MODEL FACADE AS API. From them, a DOMAIN MODEL with two SERVICES, five ENTITIES, and five VALUE OBJECTS is derived.
- Dugalic [7] discusses a purchase order management system with a Shipping and Order BOUNDED CONTEXTS. The two BOUNDED CONTEXTS are exposed to the API and use various AGGREGATES for their realization. That is a selection of domain model elements is made that are exposed as one API per BOUNDED CONTEXT, meaning that the *Expose Each Bounded Context as its Own API* variant of DOMAIN MODEL FACADE AS API is chosen. Each API offers two endpoints, one command and one query endpoint, to realize the COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) pattern [26], two times: two endpoints for a RESTful and two for a gRPC-based API. Lastly, the APIs each offer one gRPC-based PUBLISH-SUBSCRIBE endpoint for exchanging events, e.g. between the services realizing the endpoints.

5.3 Pattern: Aggregate Roots as API Endpoints

Alias. Aggregate Root Wrappers, Aggregate-oriented API Endpoints.

Context. In a software development project, you use DDD to design your DOMAIN MODEL and want to expose some parts of the software as an API.

Problem. How to derive API endpoints from domain model elements?

Forces.

- *Avoid Exposing Domain Model Details in API*: Exposing details of the domain models in the API that are not necessarily needed by at least some API consumers increases the API complexity and coupling between clients and server. This can lead to maintainability and modifiability problems. But avoiding domain model details in the API requires more intricate API design and consequently implementation, which leads to more design and implementation effort required for the initial API.
- *Data Consistency*: The API shall be designed in a way so that it is easy to maintain strict or eventual data consistency[7] as required in the given business domain context. If parts that need to be kept consistent are split across multiple API endpoints, distributed measures for ensuring consistency have to be used, which are more complex and error-prone, and offer worse performance than local consistency measures. However, a good design for data consistency is hard, especially in distributed setting. This can be avoid by co-locating data elements in the same service.
- *Chatty APIs*: If (almost) every step in the processing requires some distributed interaction, the application is much slower than a coarser design, where unnecessary distributed calls are avoided. Such APIs are too chatty. The same can happen also for the services in the backend (i.e. chatty microservices), but in this force many client-API interactions are meant that can be observed at the API surface. Also, API complexity rises because complex distributed interaction patterns need to be understood. This influences maintainability negatively, too. For example, often the client has to maintain state and orchestrate interactions between many chatty API calls.
- *Performance and Scalability*: APIs should perform as well as possible and be as scalable as required. Bad API endpoint choices, e.g. as discussed for chatty APIs and issues in data consistency, can lead to insufficient performance and scalability. Performance and scalability can be improved, if the domain object that are needed to sent back a particular response are kept together in one and the same API service where possible.
- *API Complexity*: A very fragmented or too detailed derivation of API elements from domain model elements, as well as chatty APIs, can lead to high complexity of the API, which should be avoided. For instance, fragmented or chatty APIs usually make many assumptions on complex interaction patterns or state to be kept between API calls (on client side), making the APIs hard to understand and use. Conversation patterns describing such interactions have been mined [16].
- *Coupling of API Consumer and Supplier*: A very fragmented or too detailed derivation of API elements from domain model elements, can lead to a high coupling of API consumer and supplier. As a consequence, every change in the API interface can lead to (maybe complex) changes in many clients, making it progressively harder to evolve the API.
- *Security and Privacy*: As in the DOMAIN MODEL FACADE AS API pattern, security and privacy related aspects play a role, such as avoiding excessive data exposure, enabling auditability, and supporting monitoring/observability. For example, the OWASP API Security project[8] has identified a top 10 list of API security risks, especially excessive data exposure (the number 3 on the list) is relevant for this pattern. One of the recommendations is to "review all API responses and adapt them to match what the API consumers really need." This advice is in line with the position taken in this pattern and the related patterns and practices: a domain model element should never be fully exposed in a pass-through manner just because this is technically feasible; the API designers should rather provide views and facades on the required parts of the domain model in the API.

Most of the forces discussed above are an issue for an initial design of an API and of its API clients, but become more and more problematic as the API evolves and thus needs to be maintained

[7] Eventual consistency describes a weak consistency relation which requires that all replicas of an object (here: microservices) will only eventually reach the same correct value.

[8] See https://owasp.org/www-project-api-security/, https://apisecurity.io/encyclopedia/content/owasp/api3-excessive-data-exposure

[24]. Thus finding good solutions for the other forces is essential for supporting the *Maintainability of API and API Consumers*, too.

Solution. Expose selected AGGREGATE Roots as API endpoints. Mark those elements clearly as domain model elements being exposed as API endpoints.

Solution Details. As an AGGREGATE abstracts the implementation details of a number of related ENTITIES, VALUE OBJECTS, SERVICES, and other domain model elements, it naturally serves as an interface element that can be exposed to an API as an API endpoint.

An AGGREGATE is a cluster of domain model elements such as ENTITIES, VALUE OBJECTS, SERVICES, and so on. It usually contains a root which is one of those elements and which is designated in a model as the aggregate root.

Please note that the term "exposed to" can mean that the AGGREGATE interface is rather literally mapped to the API. It can, however, also mean that a representation of the AGGREGATE is provided in the API which contain some changes compared to what is in the DOMAIN MODEL (than rather a REMOTE FACADE [10], maybe as part of a SERVICE LAYER, to the AGGREGATE is offered). For example, different operations or data types can be used in the API representation, if needed. Consider an endpoint offers the possibility to provide a WISH LIST [31] for some API operations. WISH LIST means an API client can provide in the request an enumeration of all desired data elements of the requested endpoint. This can be used to optimize resource usage in the distributed setting for operations that send possibly large amounts of data. That is, in addition to the respective domain operations, additional operations and/or data types can be used to describe the WISH LIST requests which are only present in the API but not in the DOMAIN MODEL.

To be able to apply this pattern, a DOMAIN MODEL must contain suitable AGGREGATES in the first place. If this is not the case and it is perceived as an issue of the DOMAIN MODEL design, one option is to consider redesigning parts of the DOMAIN MODEL. If the DOMAIN MODEL design is of good quality, it might be better to look out for alternative DOMAIN MODEL elements which can be abstracted into API elements. That is, AGGREGATE Roots are a good starting point for API endpoints, but some other options exist: For instance, PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives that often lead to very good results. In any case, usually deliberate, incremental design is required to find good API endpoints.

Example. To illustrate the pattern let us consider an excerpt of a DDD model of a publication management system [41]. Firstly, in Figure 5 a CONTEXT MAP is shown that describes the relations of two BOUNDED CONTEXTS, one for a *Reference Management Service* and one for a *Reference Management Frontend* part of the system. The *Reference Management Service* context's details are shown in Figure 9. It contains an AGGREGATE for paper archiving which is composed of a SERVICE for *Paper Archiving*, a *Paper Collection* ENTITY, a *Paper Item* ENTITY, and a *Paper Item Key* VALUE OBJECT.

In Figure 10, the *Paper Archive Facade* is exposed to the API as an API endpoint. This is clearly marked in the figure through the *«exposed to API as»* relation to an API endpoint. Please note that for each domain model element that is exposed as selection must be made what actually is exposed. For example, of course, not all attributes of the model element have to be exposed. This means that all members of the *Paper Archive Facade*, which here have an *«exposed to API as»* relation to an *API Element*, are offered as part of this API endpoint. That is, here the API elements *Paper Item DTO* and *Paper Item Key Data Type* are offered as elements of the API in this endpoint.

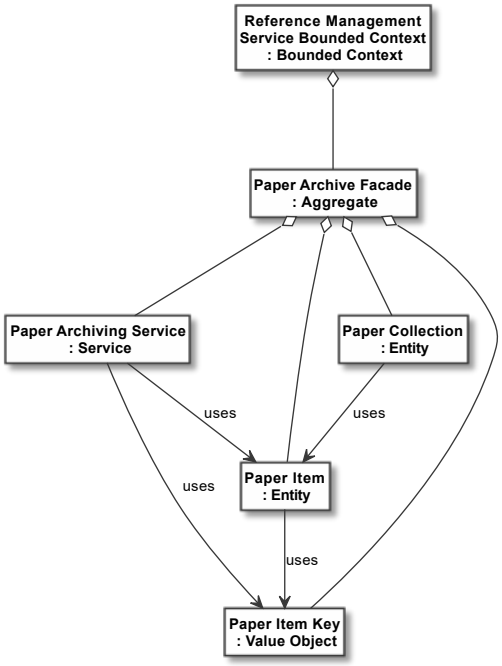

Figure 9: Publication Management Example (adapted from [41])

Pattern Variants and Alternative Practice. PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives to AGGREGATE ROOTS AS API ENDPOINTS (see [29]). They can also be seen as variants of this pattern.

Two alternative practices to this pattern and its related patterns are *Entities as API Endpoints* or *Bounded Context as API Endpoints* as shown in Figure 11. *Entities as API Endpoints* means that ENTITIES are offered as API endpoints. They, however, are often too fine-grained in the sense that then simple, shallow ENTITIES with CRUD (Create, Read, Update, Delete) operations are offered in the API. This can lead to the *Data Class* bad smell between distributed API endpoints. According to our practitioner sources, *Entities as API Endpoints* can lead to problems related to API complexity, data consistency, chatty APIs, performance and scalability isses, API understandability, API maintainability, and API evolvability. On the other hand, in some cases, *Entities as API Endpoints* can also lead to well-fitting designs where similar structures are used intentionally (e.g., the use of ENTITIES does not lead to the *Data Class* bad smell but the structurally identical DATA TRANSFER OBJECT pattern [10]). Again, careful and deliberate design is needed to come up with a well-fitting API design.

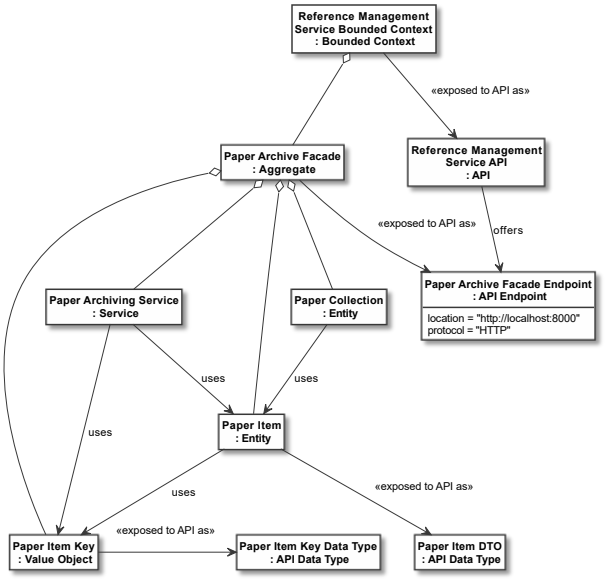

Figure 10: Publication Management Example Exposed to the API

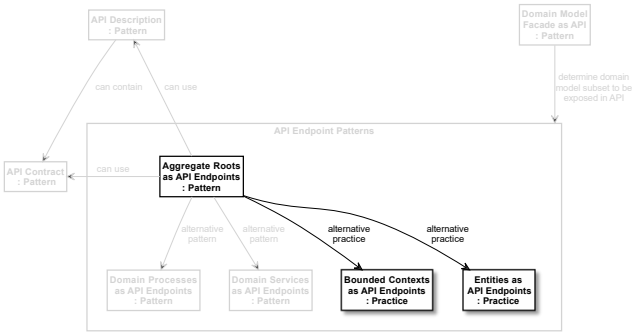

Figure 11: Aggregate Roots as API Endpoints – Alternative Practices

Instead of an AGGREGATE, *Bounded Context as API Endpoints* is a practice that offers BOUNDED CONTEXTS as composite units offered as API endpoints. But this often is much too coarse-grained leading to too large and complex API endpoints. Again, this is not true for every existing BOUNDED CONTEXT, and if careful and deliberate design uses *Bounded Context as API Endpoints*, it might produce well-designed API endpoints.

Consequences.

+ *Level of Domain Model Details Exposed:* The pattern usually provides good results regarding the amount and level of domain model details exposed in the API. AGGREGATES are coarse-grained structures that abstract the implementation details of a number of related ENTITIES, VALUE OBJECTS, SERVICES, and other domain model elements. Thus the AGGREGATES often works well as an endpoint (as also discussed in the small aggregates rule by Vernon [35]), whereas selected members can be mapped to or be represented by API elements, and other members are hidden from the API.
+ *Avoiding Data Consistency Issues:* AGGREGATES are often used as basic elements of data storage and thus it is advised that transactions should not cross AGGREGATE boundaries [9]. Thus using them in this way as API endpoints means that all "local" data consistency issues are handled by the local transactions performed within the AGGREGATE boundary. This avoids unnecessary distributed data consistency handling.
+ *Performance and Scalability:* Coarser-grained structures enable less *chatty APIs* and thus better performance and scalability than more fragmented structures, where more distributed interactions are required. The Microservice API Patterns offer patterns that might optimize API designs in such situations. For example, the REQUEST BUNDLE [46][10] pattern defines a data container that assembles multiple individual requests in a single request message. The WISH LIST pattern discussed above is another option.
+ *API Complexity and Coupling:* Coarser-grained structures that abstract details and interactions, such as AGGREGATES, reduce API complexity and coupling of API consumer and supplier.
+ *Security and Privacy:* This pattern creates a clear mapping between domain model elements and API endpoints, making it easier to trace which domain model elements at a detailed design level are exposed via the API. This can help to avoid excessive exposure of data in domain model elements, as well as fulfill auditability and/or monitoring requirements e.g. for possibly confidential parts of the domain model.
– *Better Suited Domain Model Elements Might Exist:* In some cases the AGGREGATE is not the optimal structure to represent the endpoint. For example, in a AGGREGATE that contains multiple SERVICES or domain processes, sometimes the AGGREGATE is too coarse-grained, and the SERVICES or domain processes are better suited as endpoints. In rare cases, "lonely" ENTITIES exist in domain models that make little sense to be included in any of the existing AGGREGATES. Then it might make sense to expose them directly as API endpoints.
– *Issues when Aggregate Boundaries Have to be Crossed:* Some benefits above work especially well as long as the API consumer only depends on one AGGREGATE. If transaction boundaries need to be crossed across AGGREGATES and/or chatty APIs with multiple involved AGGREGATES arise, the chosen AGGREGATE design might not yet be optimal for the purpose of exposing an API. Please note that in general crossing transaction boundaries across AGGREGATES is considered a bad practice in DDD-based design [35][11].

Related Patterns. PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives to AGGREGATE ROOTS AS API ENDPOINTS that often lead to very good results, according to the practitioner sources in our prior study (see [29]). Less often the practices *Entities as API Endpoints* or *Bounded Context as API Endpoints*, described above may lead to

[9] See e.g. https://martinfowler.com/bliki/DDD_Aggregate.html.
[10] See https://microservice-api-patterns.org/patterns/quality/dataTransferParsimony/RequestBundle.html
[11] See also: https://socadk.github.io/design-practice-repository/activities/DPR-TacticDDD.html

acceptable results, according to the practitioner sources in our prior study (see [29]) as well.

AGGREGATE ROOTS AS API ENDPOINTS (and in the same manner all alternative patterns and practices including PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS) implies to expose API endpoints as *Application Services* [35]. Vernon [35] (and also Evans [8]) distinguishes *Application Services* from *Domain Services*, i.e. services modeled as domain model elements in the DOMAIN MODEL. The set of those *Application Services* exposed together by an application form a SERVICE LAYER [10]. The SERVICE LAYER defines an application's boundary as an intermediate layer exposing a local, consumer-driven API. The API defines the client-visible interfaces of a subset of these services. For example, an application might define five *Application Services* in its SERVICE LAYER, each with its own endpoint. It might further define two APIs, one for ordinary API consumers and one an administration API; the first API exposes three of the *Application Services* as endpoints, the second one the remaining two. The DOMAIN MODEL FACADE AS API pattern describes how to establish one or more APIs based on the domain model elements to be exposed to an API through such *Application Services*.

AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices) can use API CONTRACT or API DESCRIPTION to specify the mapped API design as well as the API endpoints formally.

As discussed above, the Microservice API Patterns [46], such as WISH LIST or REQUEST BUNDLE, can help to optimize the API representation in AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices), for example, to improve performance and scalability, as well as avoiding chatty APIs.

Brown et al. [4] suggest to derive API endpoints from ENTITIES and AGGREGATES using the ENDPOINT API pattern [6], i.e., a plain mapping to endpoints following RESTful design principles. Domain processes, in contrast, are covered by the PROCESS API PATTERN, where processes are "nounified" to be mapped to RESTful resources. In all these cases, a mapping to an API endpoint is performed.

Known Uses.

- The publication management system [41] used as an example above uses the pattern to select which element makes up the *Paper Archive Facade* endpoint, as shown in Figure 10.
- Dugalic [7] discusses a purchase order management system with a Shipping and Order BOUNDED CONTEXTS, which both contain various AGGREGATES. Both BOUNDED CONTEXTS contain "main" AGGREGATES also called Shipping and Order. Each of those is exposed to two endpoints, one command and one query endpoint, to realize the COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) pattern [26], two times: two endpoints for a RESTful and two for a gRPC-based API. Lastly, the AGGREGATES each offer one gRPC-based PUBLISH-SUBSCRIBE endpoint for exchanging events, e.g. between the services realizing the endpoints.
- The Eventuate Tram Customers and Orders system[12] exposes some AGGREGATES to an event-driven API. Again, the CQRS pattern is used in the API design, too. Event handler and publisher abstractions are realized in a service that is a wrapper for the AGGREGATES.

[12]https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders

5.4 Pattern: Domain Services as API Endpoints

We present this pattern here as an extension to AGGREGATE ROOTS AS API ENDPOINTS, explaining only the differences, as the patterns are very similar. In some cases, it might make more sense to consider DDD SERVICES instead of AGGREGATE roots as the foundation for the API endpoints. For example, if one AGGREGATE naturally is composed of multiple services, and no transaction that crosses services boundaries is found in the AGGREGATE, it might be an option to expose each of the services as an endpoint in the API. This makes for instance sense, if the AGGREGATE root based API endpoints are getting very large, and a split based on the SERVICES contained in them is beneficial for API complexity and comprehensibility. In some domain models, designers deliberately use SERVICES for larger decomposition units and not AGGREGATES. Of course, one option is to redesign the model with AGGREGATES, another one is to simply use those modeled SERVICES as basis for API endpoints.

Please note that usually in DDD AGGREGATES and SERVICES are not seen as two alternative concepts, they rather complement each other. For example, a SERVICE can be the aggregate root in an AGGREGATE. However, we document this pattern here explicitly, as our empirical data indicates that practitioners sometimes only model domain services to derive APIs from them, rather than using the AGGREGATES pattern as well.

If SERVICES are used in this way, as in the cases explained in the previous paragraph, are used as decomposition units very similar to AGGREGATES, the consequences of this pattern are more or less the same as those explained in AGGREGATE ROOTS AS API ENDPOINTS.

For instance, to model the DOMAIN SERVICES AS API ENDPOINTS in the example in Figure 10 the *Paper Archiving Service* could have the *«exposed to API as»* relation to the API endpoint instead of the *Paper Archive Facade*. As the two actually exposed domain model elements are used (only) by this SERVICE, likely the same or a very similar endpoint would be realized based on this changed model. However, please note that there is a difference: The *Paper Collection* entity that is used only in the backend is part of the AGGREGATE but not used by the SERVICE. That is, the mapping of this service might lead to problems with transaction boundaries in the future, if *Paper Collection* is part of vital transactions, but it is not visible from the model as it is part of the SERVICE based API endpoint. That is, for the reason of documenting elements in the transactional boundaries, in this particular model, the AGGREGATE root is probably the better choice as a endpoint abstraction.

Please note again that here we discuss *Domain Services* as domain model elements being exposed to an API, as opposed to the *Application Services* (that might form a SERVICE LAYER) which are the actual implementation services realizing the API. See the discussion in the Related Patterns subsection of Section 5.3.

5.5 Pattern: Domain Processes as API Endpoints

Alias. Long-running application and domain services as API endpoints.

In a long running business process-like context, for instance claims processing in an insurance form or order management in a shopping and fulfillment logistics scenario, an API operation may realize a single business activity in a business process or even wrap

the complete execution of an entire process instance on the provider side. In DDD, the application layer manages process instances and delegates activity execution to the underlying domain layer.

We present this pattern here as an extension to AGGREGATE ROOTS AS API ENDPOINTS, explaining only the differences, as the patterns are very similar. In some cases, it might make more sense to consider domain (or business) processes instead of AGGREGATE roots as the foundation for the API endpoints. Like AGGREGATES, processes aggregate or compose a number of domain model elements, but offer a more step-wise or behavior-oriented view. Processes are especially useful, if the domain experts model and understand their domain in terms of process abstractions. As a domain process would be in DDD terms be modeled as a DDD SERVICE representing the process interface, maybe running on a process engine in the implementation, this pattern is in its consequences with regard to the API mapping almost identical to DOMAIN SERVICES AS API ENDPOINTS. It is introduced here as a separate pattern, as this might not be obvious to stakeholders who are used to model with process abstractions, as DDD and business process modeling are two different modeling approaches with overlaps.

In the Microservice API Patterns Language, State Creation Operations and State Transition Operations in Processing Resources model these API capabilities and responsibilities (see [42]).

The Process-Driven SOA [14] patterns explain in detail how to map processes to services, both for services realizing the process tasks and application services used for accessing the processes. The DOMAIN PROCESSES AS API ENDPOINTS pattern can be seen as an augmentation of this pattern language making the links to DDD, on the one hand, and API design, on the other hand. Pautasso and Wilde [23] present an approach to map business processes onto RESTful push services so that business processes can be modeled and observed in a RESTful way. This is one possible way to design a RESTful API when applying the DOMAIN PROCESSES AS API ENDPOINTS pattern.

6 CONCLUSION

In this paper, we have described patterns from data sets we have created in our prior research, and linked to three existing patterns (API CONTRACT, API DESCRIPTION, and FACADE). In particular, we have mined patterns and their relations on how to formally describe APIs, how to derive APIs from DOMAIN MODELS, and how to derive API endpoints from domain model elements. One of the data sets contains 14 models of systems at present. These models served as cases used for confirming the patterns in the context of systems described, implemented, or modeled by practitioners. As future work, we plan to mine additional patterns in this context and to study metrics for detecting our patterns in existing models.

Acknowledgments. We would like to thank our shepherd Filipe Correia for his valuable feedback on our paper.

The work of Cesare Pautasso and Uwe Zdun was supported by the API-ACE project, funded by SNF project 184692 and FWF (Austrian Science Fund) project I 4268. The work of Olaf Zimmermann is partially funded by the Hasler Foundation (DD-DSE, QDAR).

REFERENCES

[1] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. 2021. Facing the Giant: a Grounded Theory Study of Decision-Making in Microservices Migrations. arXiv:cs.SE/2104.00390

[2] Eric Bouwers, Joost Visser, Carola Lilienthal, and Arie van Deursen. 2010. A Cognitive Model for Software Architecture Complexity. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, Washington, DC, USA, 152–155. https://doi.org/10.1109/ICPC.2010.28

[3] Antonio Brogi, Davide Neri, Jacopo Soldani, and Olaf Zimmermann. 2019. Design principles, architectural smells and refactorings for microservices: A multivocal review. *CoRR* abs/1906.01553 (2019), 3–15. http://arxiv.org/abs/1906.01553

[4] Kyle Brown, Cees De Groot, and Chris Hay. 2019. Clound Adoption Patterns: A set of Patterns for Developers and Architects Building for the cloud. https://kgb1001001.github.io/cloudadoptionpatterns/Cloud-Native-Architecture/.

[5] Juliet Corbin and Anselm L. Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology* 13 (1990), 3–20. Issue 1.

[6] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, New York, NY, USA.

[7] Ivan Dugalic. 2019. A pattern language for microservices. https://dzone.com/articles/bounded-contexts-with-axon.

[8] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, Reading, MA.

[9] Roy T Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, Irvine, CA, USA.

[10] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley, USA.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[12] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. 2019. Benefitting from the Grey Literature in Software Engineering Research. arXiv:cs.SE/1911.12038

[13] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, New York, NY.

[14] Carsten Hentrich and Uwe Zdun. 2012. *Process-Driven SOA - Patterns for Aligning Business and IT*. CRC Press, Boca Raton, Fla.

[15] Carsten Hentrich, Uwe Zdun, Vlatka Hlupic, and Fefie Dotsika. 2015. An Approach for Pattern Mining through Grounded Theory Techniques and Its Applications to Process-Driven SOA Patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program (EuroPLoP '13)*. Association for Computing Machinery, New York, NY, USA, Article 9, 16 pages. https://doi.org/10.1145/2739011.2739020

[16] Gregor Hohpe. 2006. Workshop Report: Conversation Patterns. In *The Role of Business Processes in Service Oriented Architectures (Dagstuhl Seminar Proceedings)*, Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst (Eds.), Vol. 06291. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[17] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., USA.

[18] Stefan Kapferer and Olaf Zimmermann. 2020. Domain-driven Service Design - Context Modeling, Model Refactoring and Contract Generation. In *Proc. of the 14th Advanced Summer School on Service-Oriented Computing (SummerSOC'20) (to appear)*. Springer International Publishing, Cham, 189–208.

[19] Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, USA.

[20] Li Li and Wu Chou. 2010. Design patterns for restful communication web services. In *2010 IEEE International Conference on Web Services*. IEEE, IEEE, Washington, DC, USA, 512–519.

[21] Li Li, Wu Chou, Wei Zhou, and Min Luo. 2016. Design patterns and extensibility of REST API for networking applications. *IEEE Transactions on Network and Service Management* 13, 1 (2016), 154–167.

[22] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLop '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 24 pages. https://doi.org/10.1145/3361149.3361164

[23] Cesare Pautasso and Erik Wilde. 2011. Push-Enabling RESTful Business Processes. In *Service-Oriented Computing*, Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–46.

[24] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 2: Service Integration and Sustainability. *IEEE Software* 34, 2 (2017), 97–104. https://doi.org/10.1109/MS.2017.56

[25] Austen Rainer and Ashley Williams. 2019. Using blog-like documents to investigate software practice: Benefits, challenges, and research directions. *Journal of*

Software: Evolution and Process 31, 11 (2019).

[26] Chris Richardson. 2017. A pattern language for microservices. http://microservices.io/patterns/index.html.

[27] Dirk Riehle, Nikolay Harutyunyan, and Ann Barcomb. 2021. Pattern Discovery and Validation Using Scientific Research Methods. arXiv:cs.AI/2107.06065

[28] Apitchaka Singjai, Georg Simhandl, and Uwe Zdun. 2021. On the Practitioners' Understanding of Coupling Smells – A Grey Literature Based Grounded-Theory Study. *Accepted for publication in Information and Software Technology* 134 (2021), 106539.

[29] Apitchaka Singjai, Uwe Zdun, and Olaf Zimmermann. 2021. Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory. In *18th IEEE International Conference on Software Architecture (ICSA 2021)*. IEEE, IEEE, Washington, DC, USA.

[30] T. Sousa, H. Ferreira, and F. Correia. 5555. A Survey on the Adoption of Patterns for Engineering Software for the Cloud. *IEEE Transactions on Software Engineering* (jan 5555), 1–1. https://doi.org/10.1109/TSE.2021.3052177

[31] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. 2018. Interface Quality Patterns: Communicating and Improving the Quality of Microservices APIs. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3282308.3282319

[32] Jeffrey Stylos and Brad Myers. 2007. Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, IEEE, Washington, DC, USA, 50–60.

[33] Rasmus Svensson, Adell Tatrous, and Francis Palma. 2020. Defining Design Patterns for IoT APIs. In *European Conference on Software Architecture*. Springer, Springer International Publishing, Cham, 443–458.

[34] Davide Taibi and Valentina Lenarduzzi. 2018. On the definition of microservice bad smells. *IEEE software* 35, 3 (2018), 56–62.

[35] Vaughn Vernon. 2013. *Implementing Domain-Driven Design*. Addison-Wesley Professional, Boston, USA.

[36] Markus Voelter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, Hoboken, NJ, USA.

[37] Mark Wilkinson, Benjamin Vandervalk, and Luke McCarthy. 2011. The Semantic Automated Discovery and Integration (SADI) web service design-pattern, API and reference implementation. *Nature Precedings* 2 (2011), 8–8.

[38] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. 2018. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *Service-Oriented Computing*, Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu (Eds.). Springer International Publishing, Cham, 73–89.

[39] Wei Zhou, Li Li, Min Luo, and Wu Chou. 2014. REST API design patterns for SDN northbound API. In *2014 28th international conference on advanced information networking and applications workshops*. IEEE, IEEE, Washington, DC, USA, 358–365.

[40] Olaf Zimmermann. 2017. Microservices Tenets. *Computer Science-Research and Development* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[41] Olaf Zimmermann. 2020. Domain-Driven Service Design with Context Mapper and MDSL. https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html.

[42] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20)*. Association for Computing Machinery, New York, NY, USA, Article 9, 24 pages. https://doi.org/10.1145/3424771.3424822

[43] Olaf Zimmermann and Mirko Stocker. 2021. Design Practice Reference Guides and Templates to Craft Quality Software in Style. https://leanpub.com/dpr.

[44] Olaf Zimmermann, Mirko Stocker, and Stefan Kapferer. 2020. DPR Tutorial 1: API Design in an Online Shop. https://github.com/socadk/design-practice-repository/blob/master/tutorials/DPR-Tutorial1.md.

[45] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to Microservice API Patterns (MAP). *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)* 78, 4 (2020), 1–17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[46] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2021. Microservice API Patterns. https://microservice-api-patterns.org/.