Patterns on Designing API Endpoint Operations

APITCHAKA SINGJAI and UWE ZDUN, University of Vienna, Software Architecture Research Group, Austria

OLAF ZIMMERMANN and MIRKO STOCKER, OST Eastern Switzerland University of Applied Sciences, Switzerland

CESARE PAUTASSO, Software Institute, Faculty of Informatics, USI Lugano, Switzerland

Domain-driven design (DDD) is often applied when implementing microservices or communicating through APIs in distributed systems. APIs expose a published language that provides a view on entire domain models or subsets of such models. Hence, tactical DDD patterns such as Aggregate, Service, and Entity may not only structure API implementations, but also guide API specification work. In our previous work, we described endpoint-level patterns for this context. In this paper, we present three complementary patterns, namely Aggregated Domain Operation on API Endpoint, Event-Based API Endpoint Operation, and CRUD-Based API Operation. These patterns aim to derive API operations from the operations of Domain Services and Entities as well as Domain Events. We also discuss variants of these patterns, such as their combination with the patterns Command Query Responsibility Segregation (CQRS) and Publish/Subscribe. Our pattern mining work is based on a data set from an empirical study of 32 grey literature sources investigating practitioner views on deriving API designs from DDD models.

CCS Concepts: • **Software and its engineering → Software creation and management; Designing software.**

Additional Key Words and Phrases: application programming interfaces, distributed systems, domain-driven design (DDD), microservices, patterns

ACM Reference Format:

Singjai et al. 2021. Patterns on Designing API Endpoint Operations. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 28 (October 2021), 29 pages.

1. INTRODUCTION

In Domain-Driven Design (DDD) [Evans 2003; Vernon 2013] the (business) domain is placed at the center of software designing and architecting by rigorously crafting and specifying a DOMAIN MODEL [Fowler 2002]. This DOMAIN MODEL is then used to build a UBIQUITOUS LANGUAGE that enables software development teams to use domain terms throughout the development process for the software system. Evans [Evans 2003] classifies domain objects into types such as ENTITIES, VALUE OBJECTS, and SERVICES, which are then used to identify larger structures such as AGGREGATES or BOUNDED CONTEXTS.

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [Zimmermann 2017]. They are often identified based on DDD models [Newman 2015; Singjai et al. 2021b]. Microservices typically communicate via *APIs* in a loosely coupled fashion. These remote APIs can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or 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 [Zimmermann et al. 2020c; Zimmermann et al. 2021].

In this paper, we use the following definitions (from the Microservice API Patterns [Zimmermann et al. 2021; Lübke et al. 2019; Stocker et al. 2018]): "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* (also called *API consumers*). Each endpoint thus must have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), in HTTP-based SOAP, or in RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints".

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. This design issue has been addressed in our prior pattern study [Singjai et al. 2021c]. In other prior works, we investigated the interrelation of microservice API design and DDD [Singjai et al. 2021b] and DDD violations in the context of coupling smells [Singjai et al. 2021a]. These smells and violations are especially problematic when used in distributed setting, as explained in Section 3. In this work, we investigate the next step in API design: the design of API operations. Based on the data sets created in our previous work, we have mined the API operation design patterns presented in this work.

Our main contributions are three patterns on API operation design derived from DDD designs, namely AGGREGATED DOMAIN OPERATION ON API ENDPOINT, EVENT-BASED API ENDPOINT OPERATION, and CRUD-BASED API OPERATION which are alternatives for API operation design. We describe the patterns in detail along with multiple pattern variants and known uses for each of the patterns. 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. In particular, DDD is often applied in domains related to business problems and enterprise domains, but the patterns in the paper are not limited to these domains.

This article is structured as follows: First, we explain our research method in Section 2. Next, we present our motivation in Section 3. Section 5 discusses the patterns on API operation design derived from DDD designs. Then we discuss the related work in Section 6. Finally, in Section 7 we draw conclusions.

2. RESEARCH METHOD

The main knowledge sources used in this work are from our prior work [Singjai et al. 2021b]. In this work, we studied 32 practitioner sources from the grey literature (i.e., practitioner sources such as blog posts or system documentations [Garousi et al. 2019]) in depth using the Grounded Theory (GT) research method [Glaser and Strauss 1967; Corbin and Strauss 1990], 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 [Hentrich et al. 2015]. Riehle et al. [Riehle et al. 2020] 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. Our coding processes applied in this study are explained in detail in those two previous works [Singjai et al. 2021b; Singjai et al. 2021a].

Using the same research methods, we also studied coupling smells based on 48 practitioner sources [Singjai et al. 2021a]. 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 3, and contribute to the key forces and consequences of our patterns. Besides those prior works, we also considered existing patterns and pattern languages to enhance and detail our patterns.

In addition to the detailed studies, we have modeled the DDD-to-API mappings of twelve system descriptions and open source systems by practitioners in UML. We have used these system models to confirm our patterns, and we also feature them below to present known uses of the patterns.

3. MOTIVATION: RELATIONS TO ANEMIC DOMAIN MODELS AND COUPLING SMELLS

This section explains why coupling smells related DDD violations are problematic in distributed settings and APIs as a motivation for this work. One of the key *anti-patterns* discussed in the realm of DDD is the ANEMIC DOMAIN MODEL[1]. This anti-pattern describes a DOMAIN MODEL that fails to combine data with logic processing it in its realization. Such 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).

If an API is derived from such an ANEMIC DOMAIN MODEL often very basic elements of the DOMAIN MODEL, such as ENTITIES are exposed as e.g. RESTful services. This can lead to shallow API endpoints, where clients need to understand all the complexity in the backend. Transactional or data consistency boundaries between distributed 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. APIs are becoming hard to understand, maintain, or evolve (see discussion and referenced gray literature in [Singjai et al. 2021b]).

A typical symptom of these problems visible at the API operation level, which we focus on in this paper, are many simple and shallow ENTITIES or even VALUE OBJECTS, exposed as API endpoints, with CRUD (Create, Read, Update, Delete) operations only on them. This can lead to all negative consequences mentioned above. However, not every CRUD-based API endpoint is necessarily a bad design either: Some such endpoints expose rich domain abstractions well. Or the domain logic is inherently simple, meaning that the simple endpoint is the best possible design.

As it seems natural for a REST resource to refer to one or more nouns found in the DOMAIN MODEL, it is possible that, after these initial API designs are then fully specified and implemented, there is the danger that they could result in an ANEMIC DOMAIN MODEL. This happens if the search for finding rich[2] domain abstractions is replaced by simply using abstractions in the DOMAIN MODEL that are easy to realize as REST resources. If a substantial refactoring to 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.

Our prior studies show that these issues can lead to poor quality *Domain Model* design [Singjai et al. 2021a] and in consequence bad API operation designs as well. In our prior study on coupling smells [Singjai et al. 2021a][3], a number of practitioner sources discuss relations of coupling smells to issues in DOMAIN MODEL design. Let us illustrate a few of those relations with their consequences on API operation design:

> The *Data Class* bad smell describes a class that only offers data. If exposed to an API, this can lead to shallow ENTITIES only with CRUD operations on them, if a naive object to resource mapping is used. Among other issues this can lead to data consistency issues or issues regarding transactional boundaries, chatty APIs, high API complexity, and so on.

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

[2] Evans [Evans 2003] uses the term "rich DOMAIN MODEL" to express a DOMAIN MODEL which contains a rich understanding of the processes and rules of a domain. The opposite is an ANEMIC DOMAIN MODEL which represents rather a shallow understanding of the domain. Our patterns, in part, aim to helpt in reflecting such rich domain abstractions in the API.

[3] See also `https://refactoring.guru/refactoring/smells/` for definitions of code smells.

The *Feature Envy* bad smell describes a class or method that makes excessive use of a target class or its methods. If this happens for distributed API operations, this can lead to excessive distributed calls, which in turn leads 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 different API operations of two API endpoints, this can lead to similar issues as for *Feature Envy*, just across multiple operations of the endpoints.

The *Message Chain* bad smell describes designs containing a long sequence of method calls. If *Message Chains* are needed to work with API operations, then this leads to many distributed calls, which is a symptom of hard to understand, complex APIs with bad performance (so-called chatty APIs).

The *Indecent Exposure* bad smell describes a class that exposes internal detail. If this happens in an API operation, clients get to know the service's internal DOMAIN MODEL or other backend details, which they do not need for the work. This means, the API is more complex than needed and hard to understand.

Our patterns introduced below help to relate DOMAIN MODEL method designs to API operations, thus helping to expose API operations that have a meaning in the DOMAIN MODEL. In this context, the patterns help avoiding API operation designs that lead to these and similar smells.

4. PATTERN TEMPLATE

We use the following pattern template, which is derived from the so-called CANONICAL FORM[4], for presenting our patterns.

Name. The name of the pattern conveys characteristics of the solution. It also delivers the big picture of what the solution is.

Context. The context describes the conditions in which a pattern exists or occurs. It is a part of a discourse that relates the problem and the solution.

Problem. The problem statement describes the issues developers or architects might face in a design situation, and is phrased in form of a question.

Forces. The forces are mainly identified and synthesized from our previous work (see discussion above). They represent the core decision driver to decide for or against the use of the solution, also in relation to the other patterns in this work, which might serve as alternative options.

Solution. The solution provides an answer of the design issues posed in the problem statement.

Solution Details. The solution details extend the answer provided by the solution by covering various aspects of it in more detail.

Example. A source code based example is provided to illustrate the solution.

Pattern Variants. Known pattern variants are discussed. They represent alternative ways for providing the given solution to the problem.

Consequences. Consequences are addressing the resulting context. We designate them with (+) and (-) to indicate a positive or negative effect, respectively.

Related Patterns. We discuss related patterns from the literature.

Known Uses. Known uses provide evidence that the pattern is used in practice. We have modeled 12 systems as cases that contain known uses; in addition, we discuss known uses of the pattern from the literature.

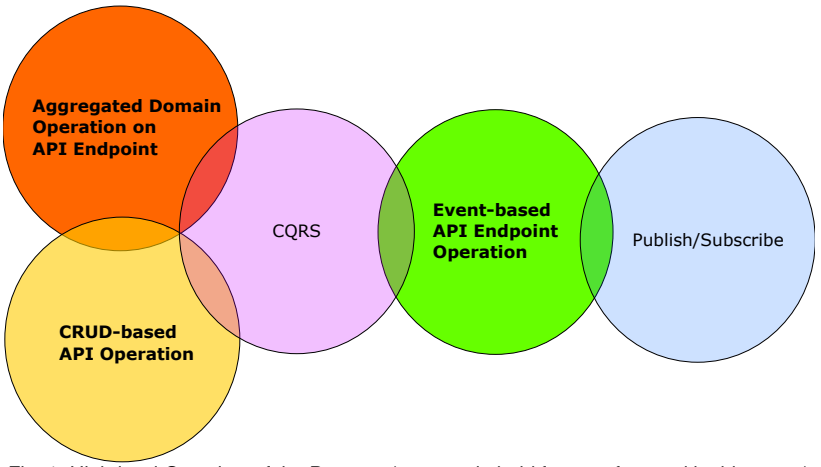

Fig. 1: High-level Overview of the Patterns (patterns in bold font are featured in this paper)

5. THE PATTERNS

The patterns presented in this section are summarized in a high-level overview in Fig. 1. The patterns introduced in this paper are highlighted in the bold-face font. They are related to two existing patterns CQRS and PUBLISH/-SUBSCRIBE [Buschmann et al. 1996]. The overlaps of the patterns define the pattern variants described below. In Fig. 2 the detailed relations of these patterns and patterns variants are shown.

The AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern describes the practice of exposing aggregated or abstracted domain operations on the API endpoints. An alternative is EVENT-BASED API ENDPOINT OPERATION which aims to expose DOMAIN EVENTS or abstracted DOMAIN EVENTS as state transition operations on the API endpoint. Alternatively, in its variant *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* DOMAIN EVENTS or abstracted DOMAIN EVENTS are offered via a PUBLISH/SUBSCRIBE [Buschmann et al. 1996] architecture or event feeds instead.

Finally, the CRUD-BASED API OPERATION pattern describes the practice to design operations on API endpoints based on the well-known *Create, Read, Update, and Delete (CRUD)* primitives. As this pattern can generate negative impact on many of its forces, when overly or wrongly applied, it shall be applied with care and only where CRUD operations are really needed by clients. Sometimes AGGREGATED DOMAIN OPERATION ON API ENDPOINT can be CRUD-based, too. Then, the variant of the two pattern *Aggregated CRUD-Based Operation on API Endpoint* can be applied.

All three patterns have a variant which combines the respective pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017]. CQRS advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. The application of CQRS usually has consequences for the API operation designs, as in the backend the views are then only eventually consistent with actions offered by the commands. That is, transactional or data consistency boundaries do not exist anymore or require further measures, and clients need to be aware of those consequences. Also, additional or other API operations might be required, e.g. API operations for compensation actions should a “transaction” fail.

All patterns described here require deriving an API endpoint from the DOMAIN MODEL first. For this we have in our prior work mined a number of patterns: AGGREGATE ROOTS AS API ENDPOINTS [Singjai et al. 2021c] starts out with AGGREGATES and their roots to derive API endpoint. DOMAIN SERVICES AS API ENDPOINTS [Singjai et al.

[4] https://wiki.c2.com/?CanonicalForm

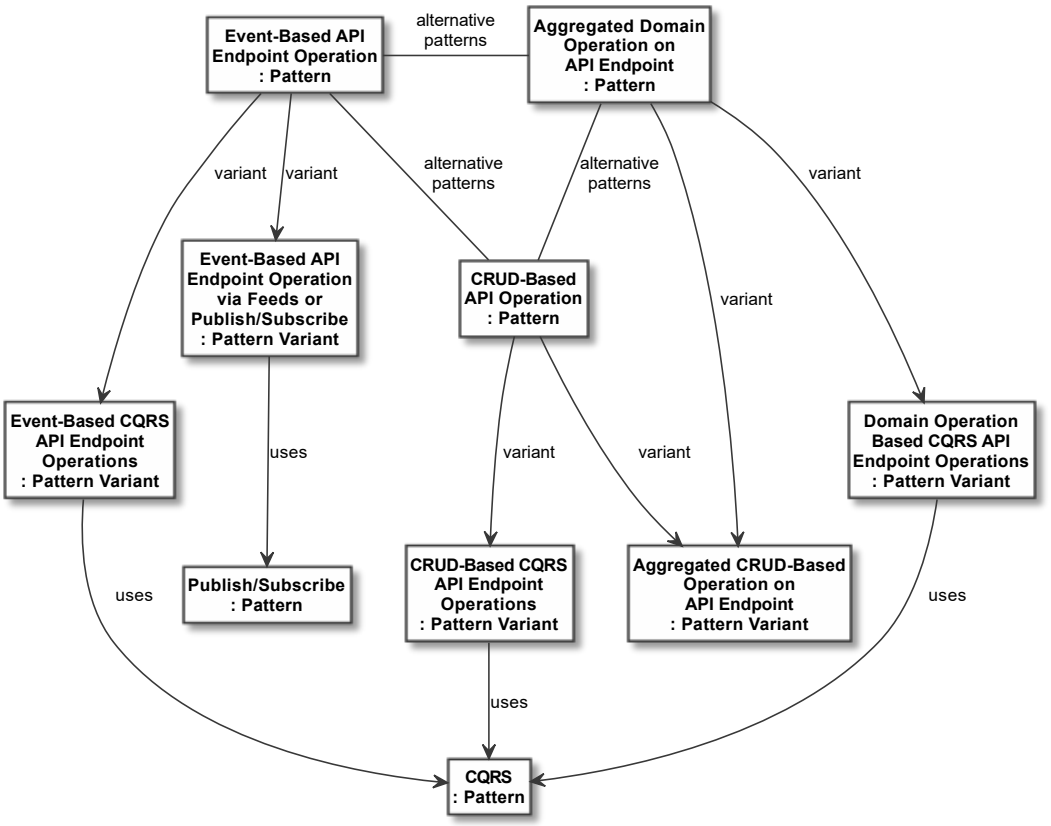

Fig. 2: Patterns and their Relations

2021c] and DOMAIN PROCESSES AS API ENDPOINTS [Singjai et al. 2021c] use domain SERVICES or processes as starting points instead. Those three options are advised most strongly by practitioners. If BOUNDED CONTEXTS or ENTITIES fit well as API boundaries, for example, they are not too large or small for the API, and cover one design concern completely, it makes sense to consider ENTITIES AS API ENDPOINTS [Singjai et al. 2021c] or BOUNDED CONTEXTS AS API ENDPOINTS [Singjai et al. 2021c], too.

5.1 Aggregated Domain Operation on API Endpoint

Context. In a software development project, you use DDD to design your DOMAIN MODEL and want to design the operations on the endpoints of an API for this project. Consider further that your DOMAIN MODEL is designed in detail, especially, it is modeling operations on the DOMAIN MODEL elements.

Problem. How to design the operations of an API endpoint in relation to the methods modeled on DOMAIN MODEL elements?

Forces

Avoid Exposing Domain Model Details in API: In the design of API operations, we want to expose the domain model details that are needed by clients to work, but no more. Any other aspects exposed might reveal more domain model details (and thus details about the backend implementation) to clients than necessary. This could e.g. lead to unnecessary coupling.

Avoid Interface Design that Limits Domain Model Design: In DDD, a DOMAIN MODEL is the core model and the basis for the UBIQUITOUS LANGUAGE of a software project. Thus, it shall not be limited in its design because of interface design considerations. For instance, consider a project starts with a simple prototype of an API operation design exposing shallow resources in CRUD style only, which sometimes happens e.g. when naively designing RESTful HTTP resource, as the HTTP verbs POST, GET, PUT, PATCH, and DELETE seems to imply this interaction style. If then the DOMAIN MODEL is designed e.g. based on those resources, it is likely that at least parts of the DOMAIN MODEL are anemic as a result.

API Understandability: At the operation level, the number and complexity of operations and the abstractions they use, influence the API understandability. This is determined by how the DOMAIN MODEL operations are mapped to API operations. Also, it is important for easing the understanding of the API by client developers that the operations are meaningful in the scope clients require. API implementation developers require a well-understandable mapping to the DOMAIN MODEL to understand the API in the context of the backend realization which is based on the DOMAIN MODEL.

Coupling of Clients to Server: The coupling of clients and servers is at the API operation level determined by the number and kinds of dependencies from clients to the operations exposed by the API. Less and less tightly coupled links or connectors (i.e., if possible, asynchronous links and indirect links such as those in the PUBLISH/SUBSCRIBE pattern [Buschmann et al. 1996]) should be preferred. This needs to decided based on the semantics of the links in the DOMAIN MODEL which are mapped to API links. For example, eventually consistent API links are usually less tightly coupled, but might make not much sense, if the DOMAIN MODEL abstractions demand a transactional relation.

Maintainability of API and API Consumers: More exposed elements of the API than necessary, more dependencies than needed, and higher coupling all can lead to negative impact on various maintainability aspects of API and API consumer, such as modular APIs, modular API clients, as well as independent testability, modifiability, extensibility, and analyzability. At the operation level, this is determined by which DOMAIN MODEL operations, links, and data elements are exposed in the API.

Chatty API, Performance, and Scalability: Too many or too fine-grained operations which need to be invoked in combination by API consumers to achieve their goals will lead to many distributed calls. If substantially more calls are exchanged than necessary, an API is called a chatty API. This should be avoided by aggregating fine-grained operations together so that the number of invocations and the length of client conversations is reduced. For example, if DOMAIN MODEL operations on AGGREGATES are exposed, instead of the operations of all members of the AGGREGATE, this usually helps to avoid chatty APIs.
More generally speaking, at API operation level, performance and scalability (in terms of requests per client, number of concurrently served requests, or computational load caused by those requests) are important forces and can be positively influenced by designing operations in such a way that unnecessary distributed calls are avoided. As discussed above, careful selection of what is exposed from the DOMAIN MODEL can help here, too, but the general impacts of design options on performance and scalability might be less obvious than just observing chatty APIs, and thus require prototyping and measurement. Also, it should be carefully considered how to expose those operations; for example, exposing many unnecessary data elements in the API operations' payload, can reduce the API provider's response behavior.
Further, the repeated calling of operations can be limited, e.g. by using patterns such as API RATE LIMITING [Zimmermann et al. 2021; Stocker et al. 2018]. API RATE LIMITING suggests to introduce rate limits per client, e.g. with the goal to avoid abusive clients being able to overload the system. All this requires careful investigation of the DOMAIN MODEL.

API Operations Reuse: Optimizing API operation design for some of the forces properties above can mean to deviate from the DOMAIN MODEL operation design. For example, by offering multiple operations for different clients, each with different performance and scalability characteristics, means to offer multiple operations per

corresponding DOMAIN MODEL operation. This might again make it hard to understand the API well in relation to the DOMAIN MODEL, but also it reduces the opportunity for reuse. There is a trade-off between reuse of operations shared by different clients and how many operations are provided overall.

Data Consistency: Operations usually process data. It is important to consider data consistency measures such as transaction boundaries and eventual consistency in the backend when designing API operations. It can be hard to manage data consistency correctly if the API consumer is responsible for ensuring data consistency guarantees. In general, all data consistency requirements stem from the DOMAIN MODEL. Here, also abstractions such as AGGREGATES are used to indicate transactional or consistency boundaries. API operation design should, if possible, adhere to those boundaries.

Solution. Primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only the DOMAIN MODEL elements required by clients and nothing more. To reach this, from the set of all domain operations modeled for the Domain Model elements to be exposed to the API, select the subset that can directly be traced back to client use cases and use this subset as a starting point for designing the API operations. Aim to offer more coarse-grained or aggregated operations on the API, rather than exposing every fine-grained operation, to avoid bloating the API with unnecessary operations which make it harder to understand and would increase coupling between API and backend implementation.

This might lead to designing API operations on coarser-grained API elements such as those representing AGGREGATES, BOUNDED CONTEXTS, or DOMAIN SERVICES, rather than those representing finer-grained API elements such as ENTITIES or VALUE OBJECTS. It might also lead to designing API operations that abstract or aggregate aspects covered in a number of detailed domain operations.

Solution Details. Our patterns on deriving API endpoints from DOMAIN MODEL elements advise us to use AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, or DOMAIN PROCESSES AS API ENDPOINTS, which are all coarse-grained DOMAIN MODEL elements abstracting from other DOMAIN MODEL elements such as ENTITIES or VALUE OBJECTS. If not of those abstractions is modeled (e.g., the model contains no AGGREGATES) or their scope does not fit well to the scope of the API to be designed (e.g., to small or to large to be easy to understand as an API), next BOUNDED CONTEXTS AS API ENDPOINTS shall be considered, which are typically yet coarser-grained structures. It is also possible to expose ENTITIES AS API ENDPOINTS but it is advised to expose them with caution because an API design where every ENTITY (or VALUE OBJECT) is exposed to the API can suffer from negative consequences. For instance, not required domain model details might be exposed through the API, which makes the API hard to understand and change, raise its complexity, introduce unnecessary coupling, and so on. This would also lead to chatty APIs, with possibly low performance and bad scalability. Those choices should be reflected at the operation level by exposing primarily operations on those coarser-grained structures. Our operation level is not only focusing on DDD and how to implement them but it is considering other architectural elements. This is why the core idea of the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern is to primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only those DOMAIN MODEL elements required by clients and nothing more.

In many protocols there is a clear-cut way how to encode the domain operations. For instance, in gRPC usually the operation would be directly encoded using the means of the protocol, typically with the same operation name as in the DOMAIN MODEL. RESTful HTTP is different here, as it usually advises us to use HTTP operations (such as POST, GET, PUT, PATCH, and DELETE) instead. Thus a mapping between domain operations and HTTP operations is required. The downsides of introducing such mappings can be an additional level of indirection, the need for maintenance of the mapping during evolution, keeping backwards compatibility, and so on. Then it might make sense to consider other implementation options such as to *encode operations as commands in the payload*. For example, this is sometimes used to encode various possible actions flexibly in one HTTP verb, e.g. actions such as `rename` are not encoded in the protocol or URL, but in the payload instead. Please note that there are also many extant APIs in which use *actions encoded in the URL* (e.g. as in "*POST /rename*"). Please note that this

practice is similar to encoding commands in the payload, but it is usually not seen as a recommended practice in RESTful HTTP. That is, RESTful HTTP guidelines often advice to use the only the HTTP verbs as actions and not encode actions in URLs.

While all such practices of circumventing RESTful conventions (encoding operations in the payload or the URL) are debatable, there are many other reasons, why encoding operations in the payload might make sense (and here encoding in the payload usually makes indeed more sense than encoding in the URL). For example, this practice can be used to encode a complex operation consisting possibly of many atomic operations. Consider realizing a REQUEST BUNDLE [Zimmermann et al. 2021] in which multiple requests are grouped and sent together in one request. This could be realized by encoding the commands required for the individual requests in the payload of the request bundle operation.

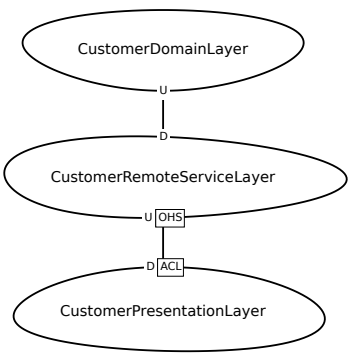

Fig. 3: Context Map for the Customer Management Example

Example from Customer Self-Service API of the Lakeside Mutual Project. The following code shows a number of operations exposed via RESTful HTTP in the *Customer Self-Service* API of the Lakeside Mutual open source project[5]. *Customer Self-Service* is an AGGREGATE which is offered as an API to clients, and, as can be seen, it aggregates various elements of the DOMAIN MODEL, such as those necessary for handling customer details or insurance quote requests. Instead of exposing detailed CRUD-based operations on each of the respective ENTITIES, the AGGREGATE-based endpoint offers only those coarse-grained domain operations which are required by clients for self-service. Please note that some of the operations perform abstracted CRUD-BASED API OPERATION; that is, those operations realize the *Aggregated CRUD-Based Operation on API Endpoint* pattern variant of this pattern.

Please note that the exposed functions realize an abstraction in the scope of the DOMAIN MODEL which is meaningful to the client ("customer self-service") as opposed to offering the detailed abstractions realizing the backend (aka the full DOMAIN MODEL).

```
export function getUserDetails(token: string): Promise<User> {
  const url = urlForEndpoint("/user")
  return getAuthenticatedJson(url, token)
}

export function getCustomer(
  token: string,
  customerId: CustomerId
): Promise<Customer> {
```

[5] https://github.com/Microservice-API-Patterns/LakesideMutual

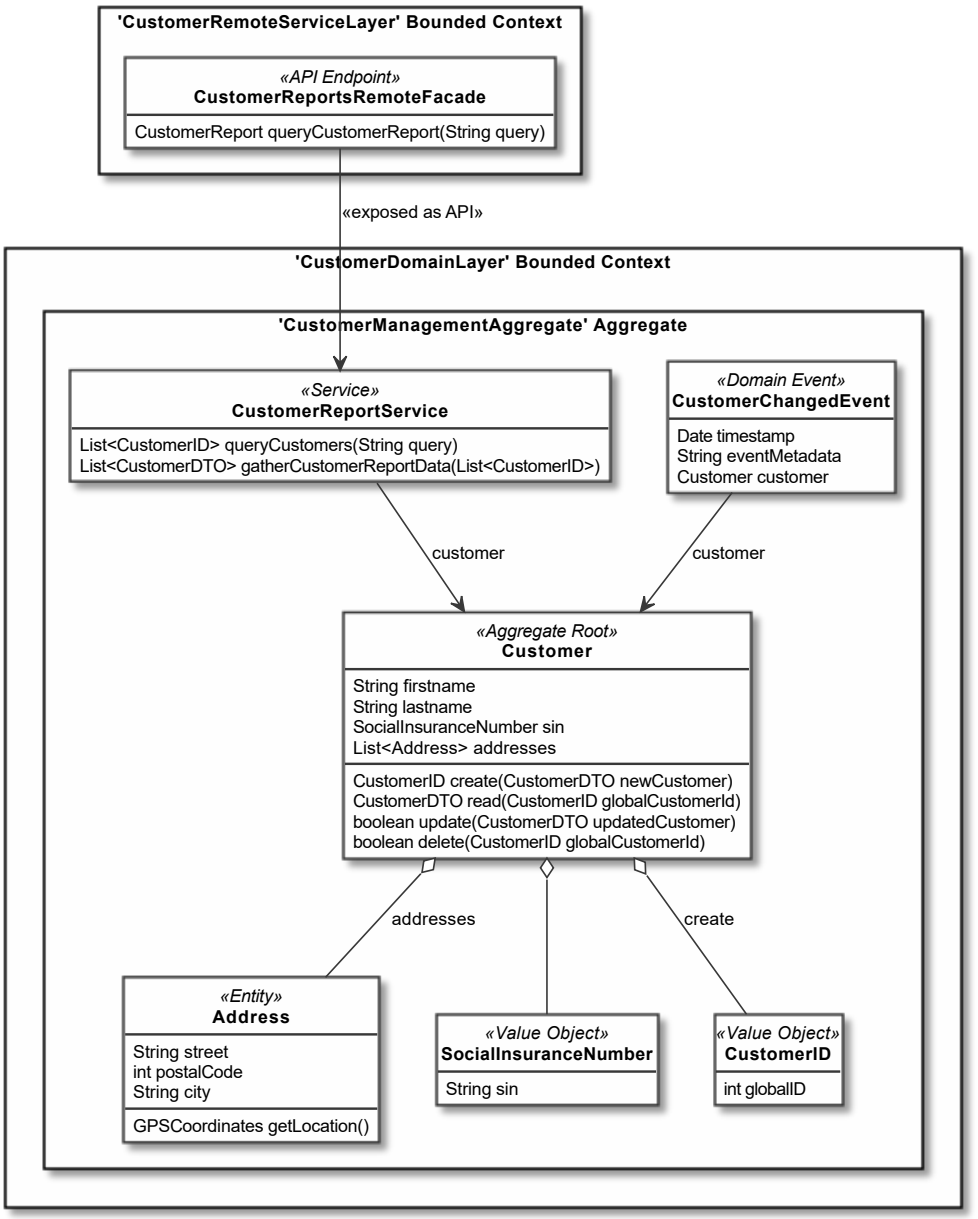

Fig. 4: Aggregated Domain API Operation for Querying Customer Reports

```
  const url = urlForEndpoint(`/customers/${customerId}`)
  return getAuthenticatedJson(url, token)
}

export function changeAddress(
  token: string,
  customer: Customer,
```

```
  address: Address
): Promise<Address> {
  // Instead of assembling the URL to change the address ourselves,
  // we use the one provided in the customer response.
  const url = customer._links["address.change"].href
  return putAuthenticatedJson(url, token, address)
}

export function completeRegistration<T>(
  token: string,
  data: T
): Promise<Customer> {
  const url = urlForEndpoint("/customers")
  return postAuthenticatedJson(url, token, data)
}

export function createInsuranceQuoteRequest(
  token: string,
  data: InsuranceQuoteRequest
): Promise<InsuranceQuoteRequest> {
  const url = urlForEndpoint("/insurance-quote-requests")
  return postAuthenticatedJson(url, token, data)
}

export function getInsuranceQuoteRequests(
  token: string,
  customerId: CustomerId
): Promise<[InsuranceQuoteRequest]> {
  const url = urlForEndpoint(`/customers/${customerId}/insurance-quote-requests`)
  return getAuthenticatedJson(url, token)
}

export function getInsuranceQuoteRequest(
  token: string,
  id: string
): Promise<InsuranceQuoteRequest> {
  const url = urlForEndpoint(`/insurance-quote-requests/${id}`)
  return getAuthenticatedJson(url, token)
}
```

Please note that in this code, helper functions such as postAuthenticatedJson are called, which handle the actual construction of the responses in a Promise:

```
export async function postAuthenticatedJson<T, U>(
  url: string,
  token: string,
  params: T
): Promise<U> {
  const response = await fetch(url, {
    method: "POST",
    headers: {
      "X-Auth-Token": token,
      "Content-Type": "application/json",
      Accept: "application/json",
    },
    body: JSON.stringify(params),
```

```
  })
  return checkStatus(response)
}
```

Pattern Variants. There are three variants of the pattern:

Aggregated CRUD-Based Operation on API Endpoint: In the CRUD-BASED API OPERATIONS pattern, we mainly discuss fine-grained CRUD-BASED API OPERATIONS. AGGREGATED DOMAIN OPERATION ON API ENDPOINT in contrast focus on the notion to offer domain operations on aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Having CRUD-BASED API OPERATIONS on those can alleviate a couple of possible risks and drawbacks mentioned in the consequences of CRUD-BASED API OPERATIONS, especially those that are related to the fine-grained nature of CRUD-BASED API OPERATIONS. The example for this pattern above contains some samples of this pattern variant. This variant is applicable when "aggregating" DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS contain CRUD-like operations in the DOMAIN MODEL.

Domain Operation Based CQRS API Endpoint Operations: The CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use aggregated domain operations, as explained in this pattern. For instance, in the example above, we would simply need to segregate the GET operations from the PUT and POST operations to create the two views. This variant is applicable when this pattern needs to be combined with CQRS.

Consequences

+ *Avoid Exposing Domain Model Details in API* : Only selected domain operations that are actually needed for client interactions are offered via the API. Other details of the DOMAIN MODEL remain hidden.
+ *Chatty API, Performance, and Scalability*: The aggregated nature of AGGREGATED DOMAIN OPERATION ON API ENDPOINT means bundling of operations and reduction of the total number of client/service interactions, which can help to avoid chatty APIs. In the same sense, they can be designed in a way optimized for performance and scalability. For example, this could be achieved by avoiding unnecessary distributed operations or introducing measures that reduce the number of requests such as REQUEST BUNDLING [Zimmermann et al. 2021] (explained briefly above). In addition to avoiding unnecessary distributed operation calls, it is possible to introduce further measures to reduce the maximum requests that need to be served at a time. For example, smaller rate limits in API RATE LIMITING [Zimmermann et al. 2021; Stocker et al. 2018] (a pattern introducing rate limits per client, e.g. with the goal to avoid abusive clients being able to overload the system) can lead to a lower number of total request that need to be served at a particular point in time.
+ *Avoid Interface Design that Limits Domain Model Design*: As explained above, this pattern tends to enable API designers to provide an abstraction in the scope of the DOMAIN MODEL which is meaningful to the client as opposed to offering the detailed abstractions realizing the backend (aka the full DOMAIN MODEL).
+ *Maintainability of API and API Consumers*: Lowering the number of possible dependencies (as suggested by this pattern in comparison to e.g. only CRUD-BASED API OPERATIONS) and thus reducing the complexity on operation level, helps to improve maintainability properties, such as modularity of the API and its consumers, as well as more independent testability, modifiability, and analyzability.
+ *Data Consistency*: Aggregated operations are often designed with transaction boundaries in mind. For example, if an operation on an AGGREGATE performs a transaction on a couple of ENTITIES in its scope, this is one data consistency issue that is solely handled in the backend and thus not a possible issue at the API level anymore.

+ *API Understandability*: This pattern tends to reduce the number of API operations (compared to using only CRUD-BASED API OPERATIONS for instance). Those API operations can be designed closer to the scope required by the clients. Those measures can improve the understandability of the API.

+/- *Coupling of Clients to Server*: On the one hand, e.g. compared to CRUD-BASED API OPERATIONS, more coarse-grained or aggregated operations can reduce number of links that introduce coupling. So overall this pattern is beneficial for coupling in many cases. On the other hand, compared to EVENT-BASED API ENDPOINT OPERATION, especially if event abstractions are offered to the client, as in the *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* variant, there is still a substantial coupling at the operation-level possible. This can especially cause problems is operations are invoked synchronously, and thus can be avoid through asynchronous operation calls to a large extent.

Related Patterns. As explained, the two patterns CRUD-BASED API OPERATIONS and EVENT-BASED API ENDPOINT OPERATION are alternatives to this pattern. Whereas CRUD-BASED API OPERATIONS is in many cases not an advisable alternative, its variant *Aggregated CRUD-Based Operation on API Endpoint* which combines CRUD-BASED API OPERATIONS with this pattern, usually is a possible option.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [Singjai et al. 2021c], DOMAIN SERVICES AS API ENDPOINTS [Singjai et al. 2021c], or DOMAIN PROCESSES AS API ENDPOINTS [Singjai et al. 2021c]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [Singjai et al. 2021c] and BOUNDED CONTEXTS AS API ENDPOINTS [Singjai et al. 2021c] practices, too.

The *encode operations as commands in the payload* implementation option can be used to realize patterns such as REQUEST BUNDLE [Zimmermann et al. 2021] which require complex specifications of the request. Such patterns can then improve performance. Scalability can be improved using patterns such as API RATE LIMITING [Zimmermann et al. 2021; Stocker et al. 2018].

The *Domain Operation Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017].

Known Uses

The *Lakeside Mutual* open source system used in the example above uses a number of AGGREGATE-based endpoints which offer domain operations which are aggregating aspects of the DOMAIN MODEL elements in their AGGREGATE scopes.

The publication management demo case for MDSL[6] uses a paper archive facade AGGREGATE which offers coarse-grained operations such as *Lookup Paper from Author*, *Create Paper Item*, and *Convert to Markdown for Website* on its aggregate root *Paper Archive Service*. Please note that one of those operations is an abstracted CRUD-BASED API OPERATION (*Create Paper Item*) which uses a concrete implementation on the *Paper Collection Backend* ENTITY.

The Cinema Microservices[7] open source system uses aggregated domain operations on DOMAIN SERVICES AS API ENDPOINTS. For instance, its *Payment* service exposes domain operations such as *makePurchase* or *getPurchaseByID* as RESTful HTTP operations as shown in Fig. 5.

The Online Shop example of the Design Practice Repository [9] exposes various AGGREGATE root SERVICE as DOMAIN SERVICES AS API ENDPOINTS. They then offer aggregate operations. For example, the *Browse and Buy*

[6] `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`

[7] `https://github.com/Crizstian/cinema-microservice`

[8] The used API tree notation: ◯ = The HTTP methods, where each method has a specific color (GET method is green, POST method is yellow, PUT method is blue, PATCH method is gray, DELETE method is red), = Path segment with no parameter, = Path segment with single parameter. For more details on the API tree diagrams see [Serbout et al. 2021].

[9] `https://github.com/socadk/design-practice-repository/blob/master/tutorials/DPR-Tutorial1.md`

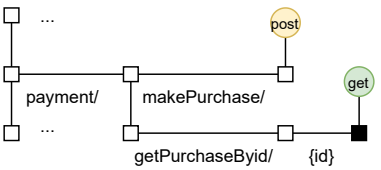

Fig. 5: API Tree[8]of Payment Endpoints in the Cinema Microservices system

service is the AGGREGATE root on the *Business to Consumer* AGGREGATE. It offers aggregate API operations such as *Create Order Item*, *Create Order*, and *Read Product Information*. Please note that all of those are abstracted CRUD-BASED API OPERATIONS which require implementations on the members of the AGGREGATE.

5.2 Pattern: Event-Based API Endpoint Operation

Context. Many microservice systems are event-based systems. Consider further that in a software development project, you use DDD to design your DOMAIN MODEL which can include DOMAIN EVENTS.

Problem. How to design the operations of an API endpoint in relation to DOMAIN EVENTS in the DOMAIN MODEL?

DOMAIN EVENTS are usually modeled as domain methods in DOMAIN MODEL, and consequently API operations in the API, that handle event emission and event reception. That is, the pattern is best applicable to design problems where the domain problem can be modeled in terms of event emissions and receptions.

Forces. This pattern has the same forces as discussed for its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT. The idea here is that the forces to be considered are essentially the same, even though they need to be considered in the context of DOMAIN EVENTS, but the consequences on them are different. For example, Loose Coupling or Avoiding Exposing Domain Model Details in APIs are forces in both cases, but the effects on the forces (aka the consequences) are fundamentally different.

Solution. Select a subset from the DOMAIN EVENTS and related technical events in the scope of an API endpoint that API clients have a need to know about, as incoming and/or outgoing events. Consider to design API-level events at a more coarse-grained or higher abstraction level than the system-internal events. For outgoing events, provide ways for clients to get notified or be able to poll for events in the API. For events incoming from the client, expose those events as STATE TRANSITION OPERATIONS[10] on the API endpoint, or provide ways with which the API can get notified about or poll for client-side events.

Solution Details. Overall, this pattern is in a number of ways similar to its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT. The main difference is that the operations exchanged as supposed to be part of an event-driven architecture. That can actually refer to different event-driven architecture patterns [Fowler 2017], but here we usually assume one or both of the following are realized by the API operations: Often event-driven architecture refers to a use of EVENT NOTIFICATIONS, i.e. “a system sends event messages to notify other systems of a change in its domain [Fowler 2017].” It can also refer to EVENT-CARRIED STATE TRANSFER, i.e. “clients of a system get updated in such a way that they do not need to contact the source system in order to do further work [Fowler 2017].” The EVENT-BASED API ENDPOINT OPERATION explains how to realize such event-driven architectures as distributed systems with the API operations realizing the incoming and outgoing event messages.

For the domain model elements in scope of your API endpoint, such as an AGGREGATE root and the scope it represents, select those DOMAIN EVENTS that should be exposed to clients. For example, in a shopping application, DOMAIN EVENTS such as “order item added to a shopping cart” or “order finished” exist and are required for clients to work properly. Thus, they should be exposed to clients as outgoing operations informing clients about system

[10] A STATE TRANSITION OPERATIONS [Zimmermann et al. 2020a; Zimmermann et al. 2021] is an operation on an API endpoint that combines client input and current state to trigger a provider-side state change f: (in,S) -> (out,S').

events or incoming operations with which clients can raise system events. In contrast, system-internal DOMAIN EVENTS not needed by clients shall not be exposed via the API. Consider for instance a client calls an event-based operation signaling that an item has been added to a shopping cart on client side. This operation is invoked on the Cart service, and the client gets a confirmation that the call was received via the used communication protocol. The Cart service, after having performed the required actions to update the cart, might raise a "shopping cart updated" event directed to other backend services. The client, who has initiated the update and thus knows of the update already (and also has a confirmation that it was received), does not need to get informed about this system-internal event.

Next, you should relate this set of exposed DOMAIN EVENTS in a state transition model and identify technical gaps in those state transitions. That is, there might be steps required for the API to work that are not yet exposed to the API. If this is the case, augment the set of exposed DOMAIN EVENTS with the additional, technical events required for the API to work. For example, before a shopping cart can be filled with order items, it might be necessary to create the cart or create a user session. Those events might not be modeled in the DOMAIN MODEL, but nonetheless they are needed to support all possible client interactions.

Example of Customer Changed Event Operations Exposed to the API. Fig. 6. extends the minimal example from Fig. 4. It uses the same DOMAIN MODEL, but highlights different operations exposed to the API: One is an event retrieval operation called *emitEvent*, and the other one is the corresponding event processor called *receiveEvent*. Both use and abstract from the *CustomerChangedEvent* domain event in the DOMAIN MODEL.

Example of Event-Based Operations in an EShop System. To illustrate the pattern consider an example from the *eShopOnContainers* open source system[11]. This system has a *UserCheckoutAccepted* DOMAIN EVENT which is triggered by a corresponding operation CheckoutAsync() on the *Basket* API, provided via an HTTP POST on the route checkout. Internally, the UserCheckoutAcceptedIntegrationEvent is raised to signal the event created by calling the API operation to other backend microservices. Integration is performed using an Event Bus based PUBLISH/SUBSCRIBE architecture [Buschmann et al. 1996].

```
[Route("checkout")]
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.Accepted)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<ActionResult> CheckoutAsync([FromBody] BasketCheckout basketCheckout,
  [FromHeader(Name = "x-requestid")] string requestId)
{
  var userId = _identityService.GetUserIdentity();

  basketCheckout.RequestId =
    (Guid.TryParse(requestId, out Guid guid) && guid != Guid.Empty) ?
    guid : basketCheckout.RequestId;

  var basket = await _repository.GetBasketAsync(userId);

  if (basket == null)
  {
    return BadRequest();
  }

  var userName = this.HttpContext.User.FindFirst(x => x.Type == ClaimTypes.Name).Value;

  var eventMessage = new UserCheckoutAcceptedIntegrationEvent(userId, userName,
```

[11] https://github.com/dotnet-architecture/eShopOnContainers

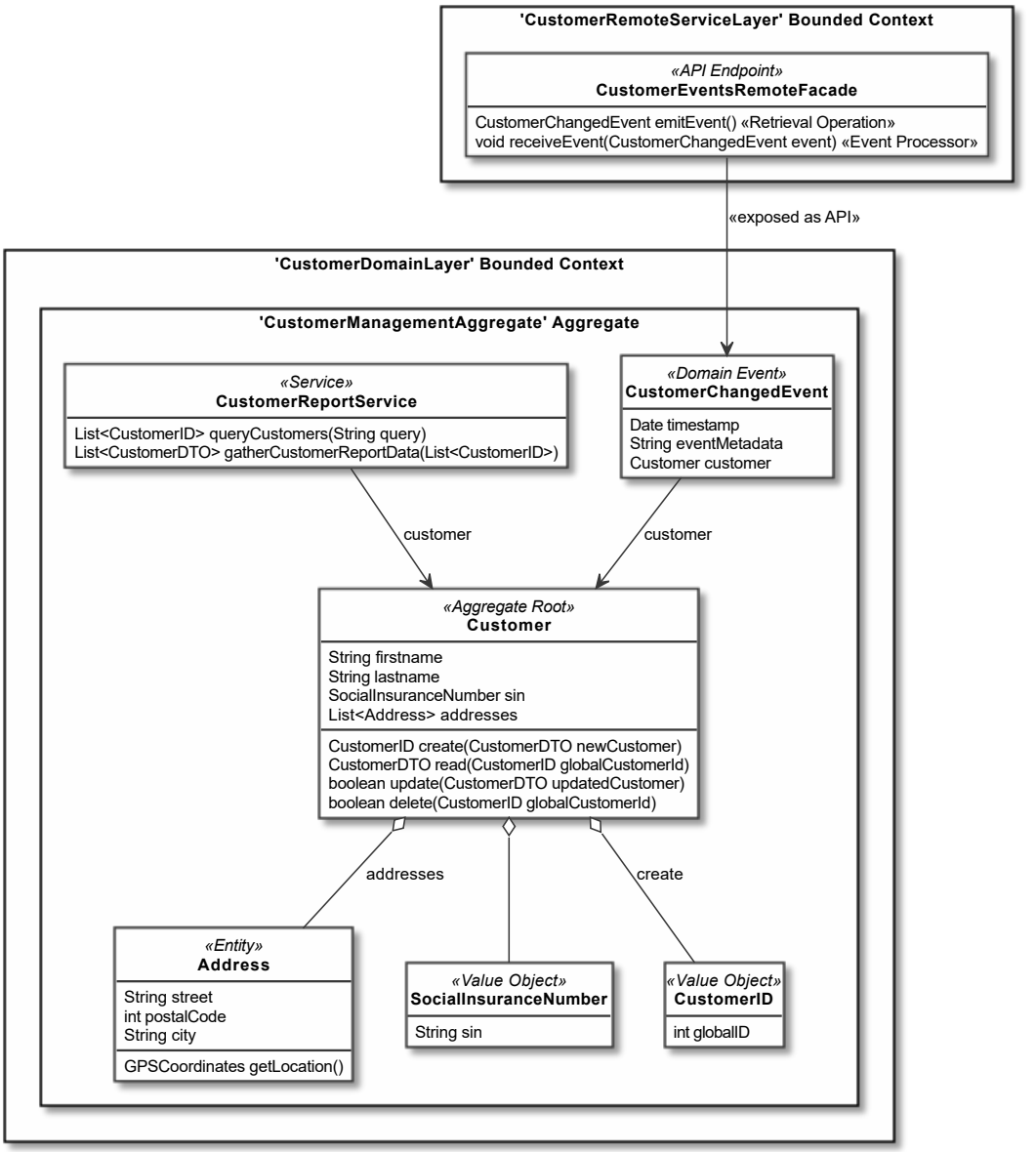

Fig. 6: Event-Based API Endpoint Operations for a Customer Changed Event

```
    basketCheckout.City, basketCheckout.Street, basketCheckout.State,
    basketCheckout.Country, basketCheckout.ZipCode, basketCheckout.CardNumber,
    basketCheckout.CardHolderName, basketCheckout.CardExpiration,
    basketCheckout.CardSecurityNumber, basketCheckout.CardTypeId,
    basketCheckout.Buyer, basketCheckout.RequestId, basket);

// Once basket is checkout, sends an integration event to
// ordering.api to convert basket to order and proceeds with
// order creation process
```

```
    try
    {
      _eventBus.Publish(eventMessage);
    }
    catch (Exception ex)
    {
      _logger.LogError(ex,
        "ERROR Publishing integration event: {IntegrationEventId} from {AppName}",
        eventMessage.Id, Program.AppName);

      throw;
    }

    return Accepted();
  }
```

Pattern Variants. There are two variants of the pattern:

Event-Based API Endpoint Operation via Feeds or Publish/Subscribe: In the example above, we have seen how a PUBLISH/SUBSCRIBE architecture can be used to signal events internally, here to other microservices realizing the API. An option is to expose the PUBLISH/SUBSCRIBE architecture or event feeds directly to clients. That is, clients publish and/or subscribe to events via the API using the internally used PUBLISH/SUBSCRIBE architecture or event feeds. If both clients and system are developed together and shall both follow an event-based architecture, this abstraction simplifies the client/system interactions as one common infrastructure is used. Very often this architecture is chosen for SOLUTION INTERNAL APIS [Zimmermann et al. 2021]. In PUBLIC APIS [Zimmermann et al. 2021] it is often not acceptable to expose the internally used infrastructure and assume clients are able or willing to use it. For example, if internally a PUBLISH/SUBSCRIBE architecture such as Apache Kafka is used, using this pattern variant would mean to expose Kafka to clients. To expose a RESTful API abstraction (maybe even on top of Kafka used in the backend) would be a solution that would not use the pattern variant, as from the viewpoint of the API, the PUBLISH/SUBSCRIBE architecture is not visible. This variant is applicable when this pattern shall be applied in a context where a PUBLISH/SUBSCRIBE architecture or event feeds are used as well.

Event-Based CQRS API Endpoint Operations: The CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use event-based abstractions, e.g. as state transition operations as explained above. Also, both can use a PUBLISH/SUBSCRIBE architecture internally to realize the eventual consistency between the services and query views. This variant is applicable when this pattern shall be applied in a context where CQRS also shall to be applied.

Consequences

+ *Avoid Exposing Domain Model Details in API* : Only selected events that are actually needed for client interactions get published as state transition operations in the API. Other details of the DOMAIN MODEL remain hidden.
+ *Chatty API, Performance, and Scalability*: Events from a number of DOMAIN MODEL elements are bundled in one state transition model to optimize the client/system interaction. This way, it is possible to avoid introducing unnecessary interactions through distributed calls between client and system exchanging the events – that would yield a chatty API. For example, defining more aggregated or higher-level events at the API level could help to optimize client/system interactions in terms of requests that have to be exchanged. More generally speaking,

such an optimized client/system interaction model can be designed for better performance and scalability. Again, additional measures than just those covering atomic operation design are possible. For example, just as discussed for AGGREGATED DOMAIN OPERATION ON API ENDPOINT, REQUEST BUNDLING [Zimmermann et al. 2021] can be applied to exchange a couple of events in one joint (bundled) distributed call. Or introducing API RATE LIMITING [Zimmermann et al. 2021; Stocker et al. 2018] can help to avoid abusive clients being able to overload the system.

\+ *Coupling of Clients to Server*: Event-based interactions usually lead to loosely coupled architectures. When exposing events as state transition operations, the clients are still coupled to those operations, but backend microservices can be loosely coupled to the receiving API microservices. In the *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* even the clients use loosely coupled dependencies.

\+ *Avoid Interface Design that Limits Domain Model Design*: In DDD, a DOMAIN MODEL is the core model and the basis for the UBIQUITOUS LANGUAGE of a software project. Thus, it shall not be limited in its design because of interface design considerations. An additional layer that abstracts the DOMAIN EVENTS, or exposing the event needed for client interactions, are both solutions to avoid a strong impact of interface design on DOMAIN MODEL design, if only selected or abstracted events are exposed.

+/- *Maintainability of API and API Consumers*: Loosely coupled relations with minimal required dependencies enable many positive maintainability properties, such as modularity of the API and its consumers, as well as more independent testability, modifiability, and analyzability. This is much harder to achieve e.g. when the fine-grained modularity implied by CRUD-BASED API OPERATIONS is needs to be maintained (i.e., many *Entities* are used as API modules), and the more strongly coupled dependencies make it harder to reach independent testability, modifiability, and analyzability. On the other hand, loose coupling often makes it more challenging to detect the impact of breaking changes: A tightly coupled system breaks immediately, a loosely coupled system may also break but will hide the root cause because of its indirection layers.

+/- *Data Consistency*: Event-based interactions often means to introduce eventual consistency [Taraporewalla 2020], but so do other distributed system interactions, especially asynchronous ones [Fowler 2015]. That is, transactions for coordination between microservices are not used and consistency issues are dealt with by compensating operations. This is more complex and substantially harder to manage than transaction-based consistency [Fowler 2015]. However, if eventual consistency needs to be embraced, the event-driven approach is a well-established one with helpful supporting patterns and practices such as EVENT NOTIFICATION, EVENT-CARRIED STATE TRANSFER, EVENT SOURCING, and CQRS [Fowler 2017].

\- *API Understandability*: Events-based interactions can be harder to understand than a simple sequence of API operations, especially if they follow a simplistic scheme as in synchronous CRUD-BASED API OPERATIONS. That is, the more asynchronous and loosely coupled an architecture is, the harder it can get to detect the impact of one operation call or a particular change. A loosely coupled and asynchronous system may hide the root cause of a defect or breaking change due to its indirections layers. Because of them, it is also hard to understand the effects of a particular API call in the system.

Related Patterns. As explained, the two patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and CRUD-BASED API OPERATIONS are alternatives to this pattern. Whereas CRUD-BASED API OPERATIONS is in many cases not an advisable alternative, its variant *Aggregated CRUD-Based Operation on API Endpoint* which combines CRUD-BASED API OPERATIONS with this AGGREGATED DOMAIN OPERATION ON API ENDPOINT, usually is a possible option.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [Singjai et al. 2021c], DOMAIN SERVICES AS API ENDPOINTS [Singjai et al. 2021c], or DOMAIN PROCESSES AS API ENDPOINTS [Singjai et al. 2021c]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [Singjai et al. 2021c] and BOUNDED CONTEXTS AS API ENDPOINTS [Singjai et al. 2021c] practices, too.

Patterns such as REQUEST BUNDLING [Zimmermann et al. 2021] or API RATE LIMITING [Zimmermann et al. 2021; Stocker et al. 2018] are options to improve performance and scalability.

The *Event-Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017].

The option *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* combines this pattern with a PUBLISH/SUBSCRIBE [Buschmann et al. 1996] architecture.

Known Uses

The *eShopOnContainers* open source system used in the example above is a system in which API operations are derived as state transition operations from DOMAIN EVENTS, and corresponding integration events are used internally to realize microservice interaction in a PUBLISH/SUBSCRIBE architecture.

Dugalic [Dugalic 2019] presents a similar exposition of API operations as state transition operations derived from *Domain Events*. In addition, each BOUNDED CONTEXT of the application is divided following the CQRS pattern into commands and queries. That is, this known use realizes the *Event-Based CQRS API Endpoint Operations* variant of the pattern. For internal inter-service communication again a PUBLISH/SUBSCRIBE architecture is used. Various implementation of this, e.g. based on the AXON Web server and RESTful HTTP, are published, too[12].

Another similar architecture is provided by the Eventuate Example application [13]. Here, DOMAIN EVENTS derived from AGGREGATES are exposed as state transition operations via CQRS command and query APIs. The Eventuate event store is used in the background as a PUBLISH/SUBSCRIBE architecture backbone. This example, additionally realized EVENT SOURCING [Richardson 2017].

The Kanban board application[14] is an example where a microservices are used via a RESTful HTTP API and WebSockets APIs. Services are based on AGGREGATES and again CQRS command and query APIs are offered. Eventual consistency is managed via an event store.

5.3 CRUD-Based API Operation

Context. In a software development project, you use DDD to design your DOMAIN MODEL and want to design the operations on the endpoints of an API for this project. Consider further that your DOMAIN MODEL is designed in detail, especially, it is modeling operations on the DOMAIN MODEL elements.

Problem. How to design the operations of an API endpoint in relation to the operations modeled on DOMAIN MODEL elements?[15]

Forces. This pattern has the same forces as discussed for its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

Solution. Design operations on API endpoints based on the well-known *Create, Read, Update, and Delete (CRUD)* primitives, which are also the basis for many primitive datastore operations abstractions, as well as the main HTTP methods (i.e., POST, GET, PUT/PATCH, and DELETE). Use this endpoint design, if DOMAIN MODEL contain only or can easily be mapped to a subset of those primitives. This includes any CRUD subset, e.g. read-only endpoints, write-only endpoints, append-only endpoints, or endpoints with no option to delete.

Limit the use of such CRUD-BASED API OPERATIONS to cases where they are essentially the only possible option. For example, they should be used, if API consumers require such Create, Read, Update, or Delete operations to function and API designers find no better-fitting, more abstract API operation.

[12] `https://github.com/idugalic/digital-restaurant`
[13] `https://github.com/cer/event-sourcing-examples`
[14] `https://github.com/eventuate-examples/es-kanban-board`
[15] Please note that the problem of this pattern is very similar to the one of AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

More design advice on this can be found in the Microservice API Patterns [Zimmermann et al. 2020a; Zimmermann et al. 2020b]. They contain several patterns that address the question of which architectural roles API endpoints play. *Information Holder Resources* are endpoints that primarily expose data management and storage operations, whereas *Processing Resources* are concerned with handling incoming action requests and commands. CRUD-BASED API OPERATIONS can typically be found on *Information Holder Resources*. It should be avoided, to design a large number of *Information Holder Resources* which are offered as distributed services and expose high semantic and operational coupling. Nygard calls this the "Entity Service Anti-Pattern"[16]. That does not mean *Information Holder Resources* should be avoided completely. Instead any use should be a conscious decision motivated and justified by the design scenario at hand to avoid negative impacts such as the coupling impact Nygard describes [Zimmermann et al. 2020b].

To reach this, ideally, some abstraction happens in the API from DOMAIN MODEL details, for example by placing the CRUD-BASED API OPERATION on an aggregating DOMAIN MODEL element, such as an AGGREGATE, domain SERVICE, or BOUNDED CONTEXT, that is exposed on the API. In rare cases, more primitive DOMAIN MODEL elements such as ENTITIES or sometimes even VALUE OBJECTS are needed as-is by API consumers. If there is no meaningful way to aggregate or abstract those DOMAIN MODEL elements further in the API, it makes sense to expose those DOMAIN MODEL elements to the API and offer CRUD-BASED API OPERATIONS on them.

Solution Details. CRUD-BASED API OPERATION offers a simple solution that is easy to design. Especially in the context of RESTful APIs where the HTTP methods support CRUD-like abstraction or in the context of APIs heavily relying on database backends, many designers start out with this pattern. If a CRUD-like abstraction is the natural abstraction to represent the DOMAIN MODEL elements in the client scope, following this pattern makes sense.

But the pattern can also be deceiving and lead to less than optimal designs. Consider a system in which each and every data element on an ENTITY and VALUE OBJECT is simply offered as CRUD-BASED API OPERATIONS. This would lead to highly complex APIs with many internal DOMAIN MODEL elements being exposed to the API that are not useful or needed in the scope of the clients. Therefore, an API consisting solely of API elements representing fine-grained DOMAIN MODEL elements, such as ENTITIES and VALUE OBJECTS, and only offers CRUD-BASED API OPERATIONS is an anti-pattern.

Thus, before applying this pattern, it shall be considered if another abstraction might be better fitting. Here, the two prime alternatives are the patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and EVENT-BASED API ENDPOINT OPERATION explained below. Please note in this context that there is also a variant of the CRUD-BASED API OPERATION pattern, explained below which offers CRUD-based operations on API elements representing aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Such *Aggregated CRUD-Based Operation on API Endpoint* are usually preferable over many CRUD-BASED API OPERATIONS on ENTITIES, as they reduce the number of necessary operations to what is actually needed by clients, as abstractions in the form needed in the client scope.

Finally, of course, some ENTITIES or other fine-grained DOMAIN MODEL elements are needed as is on client side. Then it makes no sense to introduce intermediate aggregated abstractions, but instead those ENTITIES should get exposed in the API, and if the client needs CRUD-operations on them, those should be offered as CRUD-BASED API OPERATIONS.

Our patterns on deriving API endpoints from DOMAIN MODEL elements advise us to use, if possible, AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, or DOMAIN PROCESSES AS API ENDPOINTS, which are all coarse-grained DOMAIN MODEL elements abstracting from other DOMAIN MODEL elements such as ENTITIES or VALUE OBJECTS. If this is not possible, next BOUNDED CONTEXTS AS API ENDPOINTS shall be considered, which are typically yet coarser-grained structures. It is also possible to expose ENTITIES AS API ENDPOINTS but it is advised to expose them with caution because an API design where every ENTITY or VALUE

[16] http://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/

OBJECT (which is not an AGGREGATE root) is exposed to the API suffers typically from many negative consequences. For instance, DOMAIN MODEL details might be exposed through the API, which makes the API hard to understand and change, raise its complexity, introduce unnecessary coupling, and so on. This would also lead to chatty APIs, with possibly low performance and bad scalability. Those choices should be reflected at the operation level by exposing primarily operations on those coarser-grained structures. That is, the idea of the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern is to primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only those DOMAIN MODEL elements required by clients and nothing more.

Example of RESTful Customer API Following the CRUD-based API Operation Pattern. Fig. 7. extends the minimal example from Fig. 4. It uses the same DOMAIN MODEL, but highlights different operations exposed to the API: Here, four CRUD-based operations are exposed to the API, which are mapped to the CRUD-based operations on the *Customer* ENTITY in the DOMAIN MODEL.

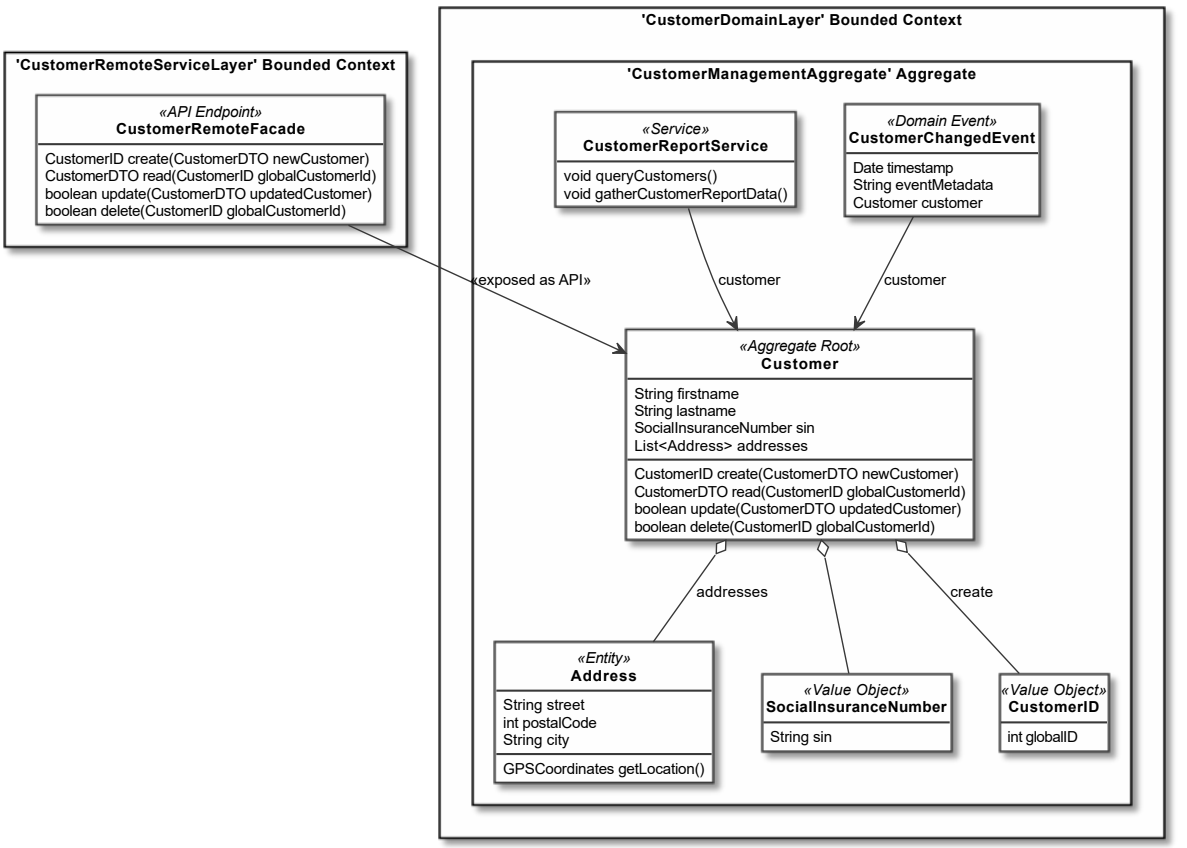

Fig. 7: CRUD-based API Operation Exposing the Customer Entity

Example of CRUD-based API Operations in the Lakeside Mutual Project. The following code shows an excerpt of the operations exposed via RESTful HTTP in the `Customer` information holder endpoint of the Lakeside Mutual open source project[17]. `Customer` is an ENTITY which is offered as an endpoint to clients. Clients such as the system's

[17] `https://github.com/Microservice-API-Patterns/LakesideMutual`

Web-based frontend pages use CRUD-BASED API OPERATIONS on it to GET, PUT, or POST customers and customer data. In this example, it makes perfect sense to expose the ENTITY with CRUD-BASED API OPERATIONS as clients such as the Web frontend page client or the customer self-service API need to perform read, update, and create operations on these data elements.

```
@RestController
@RequestMapping("/customers")
public class CustomerInformationHolder {
  ...
  @ApiOperation(value = "Get a specific set of customers.")
  @GetMapping(value = "/{ids}")
  public ResponseEntity<CustomersResponseDto> getCustomer(
  @ApiParam(value = "a comma-separated list of customer ids", required = true)
  @PathVariable String ids,
  @ApiParam(value =
  "a comma-separated list of the fields that should be included in the response",
  required = false)
  @RequestParam(value = "fields", required = false, defaultValue = "") String fields) {
    List<CustomerAggregateRoot> customers = customerService.getCustomers(ids);
    List<CustomerResponseDto> customerResponseDtos = customers.stream()
    .map(customer -> createCustomerResponseDto(customer, fields))
    .collect(Collectors.toList());
    CustomersResponseDto customersResponseDto =
    new CustomersResponseDto(customerResponseDtos);
    Link selfLink = linkTo(methodOn(CustomerInformationHolder.class)
    .getCustomer(ids, fields)).withSelfRel();
    customersResponseDto.add(selfLink);
    return ResponseEntity.ok(customersResponseDto);
  }

  @ApiOperation(value = "Update the profile of the customer with the given customer id")
  @PutMapping(value = "/{customerId}")
  public ResponseEntity<CustomerResponseDto> updateCustomer(
  @ApiParam(value = "the customer's unique id", required = true)
  @PathVariable CustomerId customerId,
  @ApiParam(value = "the customer's updated profile", required = true)
  @Valid @RequestBody CustomerProfileUpdateRequestDto requestDto) {
    final CustomerProfileEntity updatedCustomerProfile = requestDto
    .toDomainObject();

    Optional<CustomerAggregateRoot> optCustomer =
    customerService.updateCustomerProfile(customerId, updatedCustomerProfile);
    if(!optCustomer.isPresent()) {
      final String errorMessage =
      "Failed to find a customer with id '" + customerId.toString() + "'.";
      logger.info(errorMessage);
      throw new CustomerNotFoundException(errorMessage);
    }

    CustomerAggregateRoot customer = optCustomer.get();
    CustomerResponseDto response = new CustomerResponseDto(Collections.emptySet(),
    customer);
    return ResponseEntity.ok(response);
  }
  ...
```

Consequences

+/- *Avoid Exposing Domain Model Details in API* : CRUD-BASED API OPERATIONS only support selecting among the four operations CRUD offers. If those are exactly what is needed, this is optimal. In other cases, CRUD-BASED API OPERATIONS have the risk of exposing details of the DOMAIN MODEL not necessarily needed by clients.

+/- *Avoid Interface Design that Limits Domain Model Design*: This pattern leads to a good DOMAIN MODEL mapping, if the client needs the CRUD-like operations on a subset of the data elements on the DOMAIN MODEL element. Then the impact is positive; else, it can be highly negative. For instance, consider a DOMAIN MODEL element offers read-only operations of rather static information, e.g. it delivers the zip codes of a country. Then a CRUD-based DOMAIN MODEL design comes from the domain semantics and a 1:1 mapping makes sense. Consider further an early prototype of an API has been designed with only CRUD-like operations. If this design is then used as input to change DOMAIN MODEL design to "better conform" to the API, in the worst case an anemic DOMAIN MODEL is the result.

+/- *Maintainability of API and API Consumers*: If the client needs the CRUD-like operations on a subset of the data elements on the DOMAIN MODEL element, this pattern offers a positive impact on maintainability as it offers the simplest mapping possible. If this is not the case, its impact on maintainability can be rather negative, e.g. if far too many CRUD-like operations need to be maintained or they lead to complex interactions. How many CRUD-like operations are actually needed shall be determined by the semantic of the DOMAIN MODEL.

+/- *Data Consistency*: If the data consistency boundary (e.g. transaction boundary) is the DOMAIN MODEL element on which the CRUD-BASED API OPERATIONS are offered, this pattern offers very good data consistency impact. If this is not the case, in the worst case the client needs to manage consistency. Or eventual consistency measures need to be introduced in the backend. These are rather negative impacts.

+/- *API Understandability*: If CRUD-like operations are needed on the client, this pattern offers the simplest possible mapping, which is thus easy to understand. If instead far too many CRUD-like operations are offered or they lead to complex interactions, this makes the API complex and thus hard to understand.

+/- *Coupling of Clients to Server*: CRUD-BASED API OPERATIONS usually lead to a substantial level of coupling, especially compared to EVENT-BASED API ENDPOINT OPERATION, if certain consistency boundaries need to be considered. However, CRUD-BASED API OPERATIONS can also help to decouple different API clients. For example, consider some shared data among different clients, and one client creates the data, and the other one reads it and then deletes it. The two clients never meet or even need to know about each other. Thus, impact on coupling highly depends on DOMAIN MODEL semantics and how well the mapping is designed.

- *Chatty API, Performance, and Scalability*: CRUD-BASED API OPERATIONS can be very chatty, as they are among the most fine-grained API operations possible. If that is exactly what is needed by clients, they can be applied nonetheless, but still lead to chattiness. Likewise, performance and scalability are highly dependent on the number of distributed calls exchanged. They are also influenced by message payloads. Performance and scalability can be degraded if CRUD-BASED API OPERATIONS are not exactly what is needed by clients. Patterns that help to aggregate responses such as PAGINATION [Zimmermann et al. 2021] or WISH LIST [Zimmermann et al. 2021], or help to aggregate requests such as REQUEST BUNDLE [Zimmermann et al. 2021], can help to reduce these issues.

- *API Operations Reuse*: The rather fine-grained API operations that are a consequence of this pattern can quickly lead to many exposed operations and to optimizations per group of clients. As a consequence, it might be hard to reuse API operations well.

Pattern Variants. There are two variants of the pattern:

Aggregated CRUD-Based Operation on API Endpoint: We mainly discussed fine-grained CRUD-BASED API OPERATIONS above. The previously discussed pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT

discusses the notion to offer domain operations on aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Having CRUD-BASED API OPERATIONS on those can alleviate a couple of possible risks and drawbacks mentioned in the consequences of CRUD-BASED API OPERATIONS above, especially those that are related to the fine-grained nature of CRUD-BASED API OPERATIONS. The example in the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern contains some samples of this pattern variant. This variant is applicable when "aggregating" DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS contain CRUD-like operations in the DOMAIN MODEL.

CRUD-Based CQRS API Endpoint Operations: The CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use CRUD-BASED API OPERATIONS, as explained in this pattern. This then would require to segregate the Read operations from the Create, Update, and Delete operations in the Query and Command APIs. As then Read operations would be provided as only a view on the Command part of the API, data would be eventually consistent as a consequence. This can lead to different operations and interactions in the API than in a non-segregated view, e.g. additional compensation action commands might need to be added to cope with transaction issues due to eventual consistency. This variant is applicable when this pattern shall be applied in a context where CQRS also shall to be applied.

Related Patterns. As explained, the two patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and EVENT-BASED API ENDPOINT OPERATION are the prime alternatives that should be considered before considering CRUD-BASED API OPERATIONS. *Aggregated CRUD-Based Operation on API Endpoint* is a combination of CRUD-BASED API OPERATIONS with AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

Patterns that help to aggregate responses such as PAGINATION [Zimmermann et al. 2021] or WISH LIST [Zimmermann et al. 2021], or help to aggregate requests such as REQUEST BUNDLE [Zimmermann et al. 2021], can help in reducing negative performance and scalability impacts, and thus avoid chatty APIs. See the Consequences section for details.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [Singjai et al. 2021c], DOMAIN SERVICES AS API ENDPOINTS [Singjai et al. 2021c], or DOMAIN PROCESSES AS API ENDPOINTS [Singjai et al. 2021c]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [Singjai et al. 2021c] and BOUNDED CONTEXTS AS API ENDPOINTS [Singjai et al. 2021c] practices, too.

The *CRUD-Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [Richardson 2017].

Known Uses

The *Lakeside Mutual* open source system discussed above contains a number of ENTITY and AGGREGATE-based endpoints which offer CRUD-like domain operations. Thus both the main pattern variant and *Aggregated CRUD-Based Operation on API Endpoint* are supported. The example of this pattern shows the first variant, the example of AGGREGATED DOMAIN OPERATION ON API ENDPOINT shows the second variant.

The Pokemon API[18] offers many ENTITIES exposed to the API. On those it mainly offers a large collection of GET operations. Such read-only APIs in which each detail needs to be read by clients might offer some potential of aggregation and abstraction in performing multiple requests at once. Also, an API that uses a query language or interface on an AGGREGATE might be better suited. But as the fine-grained data is needed by the clients, CRUD-BASED API OPERATIONS is here not as negative as it would be on an API that would offer all four types of CRUD operations. But still it has some negative impacts on some of the forces discussed. For instance,

[18] `https://github.com/PokeAPI/pokeapi`

the API is more complex than necessary and thus harder to understand and maintain. More fine-grained calls than necessary might lead to non-optimal performance, scalability, and chatty interactions, and so on.

For similar reasons, the disease.sh open API for disease-related statistics[19] offers many GET operations on fine-grained API endpoints and is thus based on CRUD-BASED API OPERATIONS, too. As example from this API is shown in Fig. 8. It has similar negative impacts on forces as discussed for the Pokemon API.

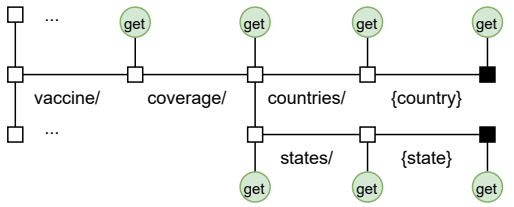

Fig. 8: The API Tree of Vaccine Endpoints in the disease.sh

The publication management demo case for MDSL[20] offers a CRUD-BASED API OPERATION *Create Paper Item* which is an instance of the *Aggregated CRUD-Based Operation on API Endpoint* variant. It requires a concrete implementation on the *Paper Collection Backend* ENTITY.

6. 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 [Evans 2003] and Vernon [Vernon 2013] describe DDD practices as patterns. Many distributed systems patterns exist, too. The closest are our Microservice API Patterns [Zimmermann et al. 2021; Lübke et al. 2019] 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 [Hohpe and Woolf 2003], Remoting Patterns [Voelter et al. 2004], Enterprise Application Architecture patterns [Fowler 2002], Cloud Adoption Patterns [Sousa et al. 2021; Brown et al. 2019], and Service Design Patterns [Daigneau 2011], 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.

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

In our prior work [Zdun et al. 2018] related to the Microservice API Patterns [Zimmermann et al. 2021; Lübke et al. 2019] we identified Architectural Design Decisions (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. The Microservice API Patterns [Zimmermann et al. 2020a; Zimmermann et al. 2020b] language features several patterns that address the question of which architectural roles API endpoints play. Information Holder Resources are endpoints that primarily expose data management and storage operations, whereas Processing Resources are concerned with handling incoming action requests and commands. The general Information Holder Resource is further refined into sub-patterns depending on the life span and mutability of the data contained.

Context Mapper [Kapferer and Zimmermann 2020] provides a Domain-Specific Language (DSL) and supporting tools for model-driven Domain-Driven Design (DDD). It focuses on modeling based on strategic and tactical DDD

[19] https://github.com/disease-sh/API

[20] https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html

Table I. : Comparison to Related Works

Work/Reference	Core Topics					Focus Area	Methodology
	API	Microservices	Smells	Refactoring	Decisions		
Zdun, Stocker, Zimmermann, Pautasso, and Lübke [Zdun et al. 2018]	Yes	Yes	No	No	Yes	ADDs for Microservice APIs	Grounded Theory (GT)
Lübke, Zimmermann, Pautasso, Zdun, and Stocker [Lübke et al. 2019]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Kapferer and Zimmermann [Kapferer and Zimmermann 2020]	Yes	Yes	No	Yes	No	Domain Driven Service Design	Empirical Validation
Zimmermann, Lübke, Zdun, Pautasso, and Stocker [Zimmermann et al. 2020a]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Zimmermann, Pautasso, Lübke, Zdun, and Stocker [Zimmermann et al. 2020b]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Taibi and Lenarduzzi [Taibi and Lenarduzzi 2018]	Yes	Yes	Yes	No	No	Microservice Bad Smells	Interviews
Stylos and Myers [Stylos and Myers 2007]	Yes	No	No	No	Yes	API design decisions, API quality attributes	Multi-vocal Literature Review
Li and Chou [Li and Chou 2010]	Yes	No	No	No	Yes	RESTful Communication Web Services	Case Study
Ayas, Leitner, and Hebig [Ayas et al. 2021]	Yes	Yes	No	No	Yes	Decision-Making in Microservices	GT/Interviews
Brogi, Neri, Soldani, and Zimmermann [Brogi et al. 2019]	Yes	Yes	Yes	Yes	No	Microservice Architectural Smells	Systematic Literature Review
Our work	Yes	Yes	Yes	No	No	DDD-based API design	Pattern Mining based on GT/Grey Literature Study

patterns such as Bounded Context, Aggregate, Entity, and Service. The domain model to API mappings presented in this paper are partially supported in Context Mapper. Platform-independent API descriptions in MDSL, another DSL, can be generated from the DDD models. These transformations map the operations in Aggregates and their Root Entities and Services to API endpoints and operations.[21] MDSL, in turn, provides mappings to microservice technologies that can be generated. These mappings include OpenAPI/Swagger interface descriptions, gRPC Protocol Buffer specifications and Jolie services (that in turn can be transformed into port types in WSDL and XML Schema). An interface refactoring catalog and tool support for refactoring API designs according to patterns are emerging as well [Stocker and Zimmermann 2021].

Taibi and Lenarduzzi [Taibi and Lenarduzzi 2018] define a number of microservice bad smells. As discussed in Section 3, 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.

Brogi et al. [Brogi et al. 2019] present a multivocal 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.

Stylos and Myers [Stylos and Myers 2007] categorized and organized API design decisions based on a multivocal literature review. They investigated the literature in API usability, whereas we mainly focus on the interrelation between API and DDD.

[21] See `https://contextmapper.org/docs/mdsl/`

Li and Chou [Li and Chou 2010] propose three design patterns for RESTful Web 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. [Ayas et al. 2021] 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.

There are number of API design patterns for specific domains, for example, northbound API of Software-Defined Networking (SDN) [Li et al. 2016; Zhou et al. 2014], Internet of Things (IoT) [Svensson et al. 2020], and biological data [Wilkinson et al. 2011]. 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, here in particular API operation design.

7. CONCLUSION

In this paper, we described patterns for deriving API operations from domain-driven design models. This work draws upon data sets we have created in our prior research. In particular, we mined patterns and their relations on how to derive AGGREGATED DOMAIN OPERATIONS ON API ENDPOINTS, API ENDPOINTS BASED ON DOMAIN EVENTS, and CRUD-BASED API OPERATION endpoints. These patterns originate from an in-depth empirical study of grey literature authored by practitioners. In our pattern mining, we have also considered twelve detailed open source system models (which we modeled from systems implemented or documented by practitioners), from which we have given examples and known uses in this paper. As future work, we plan to mine additional patterns in this context and to study metrics for detecting our patterns in existing models.

REFERENCES

Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. 2021. Facing the Giant: a Grounded Theory Study of Decision-Making in Microservices Migrations. *arXiv preprint arXiv:2104.00390* (2021).

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). `http://arxiv.org/abs/1906.01553`

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

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

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

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

Ivan Dugalic. 2019. A pattern language for microservices. `https://dzone.com/articles/bounded-contexts-with-axon`. (2019).

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

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

Martin Fowler. 2015. Microservice Trade-Offs. `https://martinfowler.com/articles/microservice-trade-offs.html`. (2015).

Martin Fowler. 2017. What do you mean by "Event-Driven"? `https://martinfowler.com/articles/201701-event-driven.html`. (2017).

Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. 2019. Benefitting from the Grey Literature in Software Engineering Research. (2019).

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

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 Pat-*

tern Languages of Program (EuroPLoP '13). Association for Computing Machinery, New York, NY, USA, Article 9, 16 pages. DOI:http://dx.doi.org/10.1145/2739011.2739020

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

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 2020) (to appear)*. Springer International Publishing, Cham, 189–208.

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.

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.

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. DOI:http://dx.doi.org/10.1145/3361149.3361164

Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly.

Chris Richardson. 2017. A pattern language for microservices. http://microservices.io/patterns/index.html. (2017).

Dirk Riehle, Nikolay Harutyunyan, and Ann Barcomb. 2020. Pattern discovery and validation using scientific research methods. https://dirkriehle.com/2020/03/05/pattern-discovery-and-validation-using-scientific-research-methods-technical-report/. (2020).

Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. 2021. From OpenAPI Fragments to API Pattern Primitives and Design Smells. In *European Conference on Pattern Languages of Programs (EuroPLoP'21)*. ACM, ACM, Virtual Kloster Irsee, Germany. DOI:http://dx.doi.org/10.1145/3489449.3489998

Apitchaka Singjai, Georg Simhandl, and Uwe Zdun. 2021a. 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.

Apitchaka Singjai, Uwe Zdun, and Olaf Zimmermann. 2021b. 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.

Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann, and Cesare Pautasso. 2021c. Patterns on Deriving APIs and API Endpoints from Domain Model Elements. In *Proceedings of the European Conference on Pattern Languages of Programs 2021 (EuroPLoP '21)*. Association for Computing Machinery, New York, NY, USA.

Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. 2021. A Survey on the Adoption of Patterns for Engineering Software for the Cloud. *IEEE Transactions on Software Engineering* (2021).

Mirko Stocker and Olaf Zimmermann. 2021. From Code Refactoring to API Refactoring: Agile Service Design and Evolution. In *Service-Oriented Computing*, Johanna Barzen (Ed.). Springer International Publishing, Cham, 174–193.

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. DOI:http://dx.doi.org/10.1145/3282308.3282319

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.

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.

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

Sarah Taraporewalla. 2020. Event Driven Architecture Terminology. https://sarahtaraporewalla.com/architecture/Event-Driven-Architecture-Terminology. (2020).

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

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.

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.

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.

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.

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

Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020a. 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. DOI:http://dx.doi.org/10.1145/3424771.3424822

Olaf Zimmermann, Cesare Pautasso, Daniel Lübke, Uwe Zdun, and Mirko Stocker. 2020b. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proceedings of the European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 25 pages. DOI:http://dx.doi.org/10.1145/3424771.3424821

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020c. 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. DOI:http://dx.doi.org/10.4230/OASIcs.Microservices.2017-2019.4

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

Received May 2021; revised September 2021; accepted September 2024