Interface Responsibility Patterns: Processing Resources and Operation Responsibilities

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

Daniel Lübke
iQuest GmbH, Hanover, Germany

Uwe Zdun
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland

Mirko Stocker
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

ABSTRACT

Remote Application Programming Interfaces (APIs), as for instance offered in microservices architectures, are used in almost any distributed system today and are thus enablers for many digitalization efforts. It is hard to design such APIs so that they are easy and effective to use; maintaining their runtime qualities while preserving backward compatibility is equally challenging. Finding well suited granularities in terms of the architectural capabilities of endpoints and the read-write semantics of their operations are particularly important design concerns. Existing pattern languages have dealt with local APIs in object-oriented programming, with remote objects, with queue-based messaging and with service-oriented computing platforms. However, patterns or equivalent guidances for the architectural design of API endpoints, operations and their request and response message structures are still missing. In this paper, we extend our microservice API pattern language (MAP) and introduce endpoint role and operation responsibility patterns, namely *Processing Resource*, *Computation Function*, *State Creation Operation*, *Retrieval Operation*, and *State Transition Operation*. Known uses and examples of the patterns are drawn from public Web APIs, as well as application development and system integration projects the authors have been involved in.

CCS CONCEPTS

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

ACM Reference Format:
Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany*. ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3424771.3424822

EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany

ACM ISBN 978-1-4503-7769-0/20/07…$15.00
https://doi.org/10.1145/3424771.3424822

1 INTRODUCTION

Microservices architectures have evolved from previous incarnations of Service-Oriented Architectures (SOAs) [18]. They consist of independently deployable, scalable and changeable services, each having a single responsibility. These responsibilities model business capabilities. Microservices are often deployed in lightweight virtualization containers, encapsulate their own state, and communicate via message-based remote APIs in a loosely coupled fashion. Microservices solutions leverage polyglot programming, polyglot persistence, as well as DevOps practices including decentralized continuous delivery and end-to-end monitoring [25], [32], [47].

Microservice API designers have to address concerns such as: How many services should be exposed? Which service cuts let services and their clients deliver user value jointly, but couple them loosely? How often do services and their clients interact to exchange data? How much and which data should be exchanged? [51] The Microservice API Patterns (MAP) at www.microservice-api-patterns.org[1] cover and organize this design space, providing guidance distilled from the collective experience of the API design community.

This particular paper deals with two specific problems of designing API endpoints and their operations (in the context of microservice architectures):

- First, we briefly investigate the design question: *Which architectural role should an API endpoint play?*
- Next, we go one level down and ask: *What is the responsibility of each API operation?*

The drivers for API introduction and requirements for API design are diverse. As a consequence, the roles that APIs play in applications and service ecosystems differ widely. Sometimes, an API client just wants to inform the provider about an incident, or hand over some data; sometimes they request provider-side data to be able to continue client-side processing. Sometimes the provider has to perform a lot of complex processing to satisfy the client's information need, sometimes it can simply return a data element that already exists as part of the server-side application state. Some of the provider-side processing, whether simple or complex, may change server-side application state, some might leave this state untouched.

[1] https://microservice-api-patterns.org/

In response to these challenges, our responsibility patterns cover two distinct main architectural roles for API endpoints: *Processing Resources* are resources whose primary function is to handle incoming action requests (also known as activities or commands), whereas *Information Holder Resources* are resources whose primary function is storage and management (including retrieval) of data or meta-data. To decide between these two, API designers have to prioritize one of two options:

- *Should data or processing be the leading design concept on the API endpoint level?*

Within this paper we cover the endpoint-level *Processing Resource* pattern first, which emphasizes processing over data. Its alternative *Information Holder Resource*, in which the designer emphasizes data over processing, is covered in another paper[2].

Next, more fine-grained decisions have to be made, covered in four patterns of widely used API *operation responsibilities*:

- *Computation Function*: An operation that computes a result solely from the client input and does not read or write server-side state.
- *State Creation Operation*: An operation that creates states on an API endpoint that is in essence write-only. Here 'in essence' means that such operations might need to read some state, e.g. to check for duplicate keys in existing data before creation, but their main purpose should be state creation.
- *Retrieval Operation*: A read-only operation that only finds and delivers data without allowing clients to change it. The data may be manipulated (for instance, aggregated) before being returned, but does not have to. Some retrieval operations search for data, others access single data elements.
- *State Transition Operation*: An operation that performs one or more activities causing a server-side state change. Examples of such operations are full and partial updates to server-side data, as well as deletions of such data.

Figure 1 shows the two the endpoint role patterns, *Processing Resource* (covered in this paper) and *Information Holder Resource* (covered in the companion paper [50]), as well as the four operation responsibility patterns (covered in this paper). The relations among the patterns are shown as well.

The remainder of this paper is structured as this. Section 2 presents related work. Section 3 provides an overview of our pattern language, its categories and patterns published so far; it also introduces the API design vocabulary used in the pattern texts as well as our pattern template. Section 4 features the five patterns. Section 5 concludes and gives an outlook.

2 RELATED WORK

2.1 Related pattern languages

We discussed pattern languages that deal with distributed system and API design in our previous three EuroPLoP papers [52], [40], [28], including Remoting Patterns [43], Enterprise Integration Patterns (EIP) [19], Fowler's patterns Service Layer and Remote Facade [11], POSA vol. 4 with its distributed systems patterns [7], service design patterns by Daigenau [8], and microservices patterns [39].

Brown and Woolf describe a set of implementation patterns for building applications using microservices [6]. They are primarily focused on the overall architecture, but also cover DevOps and scalable storage patterns. Rather surprisingly, they argue that microservices tend to be coarse-grained (despite their name) so that they can implement "clear business capabilities", a tenet that microservices share with previous generations of service-oriented architectures [47]. Their *Business Microservice*, *Adapter Microservice*, and also *Backends for Frontends* all expose remote APIs, which can be described architecturally by our patterns in terms of their request and response message structures, qualities, evolution and versioning strategies as well as endpoint roles and responsibilities. Hence, the two pattern languages complement each other nicely.

2.2 Other existing design heuristics

One can find many excellent books providing deep advice about using RESTful HTTP, e.g., which HTTP verb or method to pick to implement a particular operation, or how to apply asynchronous messaging including routing, transformation, and guaranteed delivery [1], [19]. Strategic Domain-Driven Design [9], [42] can assist with service identification. SOA, cloud and microservice infrastructure patterns have already been proposed, and structuring data storages is also well understood. Our previous publications [52], [40] and [28] cover such related works; the MAP website also gives reading recommendations[3].

2.3 Responsibility-Driven Design (RDD)

The patterns in this paper represent vastly different invocation, processing, and state management characteristics. To order and structure the design space, we adopt terminology and *role stereotypes* from *Responsibility-Driven Design (RDD)*[4]. In RDD, a stereotype is "a conventional, formulaic, and oversimplified conception, opinion, or image". An application is "a set of interacting objects", an object is "an implementation of one or more roles" (here: microservice). A role is "a set of related responsibilities", a responsibility is "an obligation to perform a task or know information". A collaboration is "an interaction of objects or roles (or both)", a contract is "an agreement outlining the terms of a collaboration" [44].

Our microservice API design terms relate to the more general RDD concepts in the following way: API operations take over a responsibility, and API and their endpoints assemble these responsibilities into roles. The collaborations then arise from calls to API operations (a.k.a. service invocations). The *API Description*, presented in a previous paper [28], specifies the contract.

All API endpoints can be seen as (remote) *interfacers* that provide and protect access to *service providers*, *controllers/coordinators*, as well as *information holders* and *structurers*. RDD defines these and other role stereotypes like this:

- An interfacer "transforms information and requests between distinct parts of a system".
- A service provider "performs work on demand".
- A controller "makes decisions and closely directs others' actions" and a coordinator "mechanically reacts to events".

[2] "Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources" [50].

[3] https://microservice-api-patterns.org/relatedPatternLanguages

[4] http://www.wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf

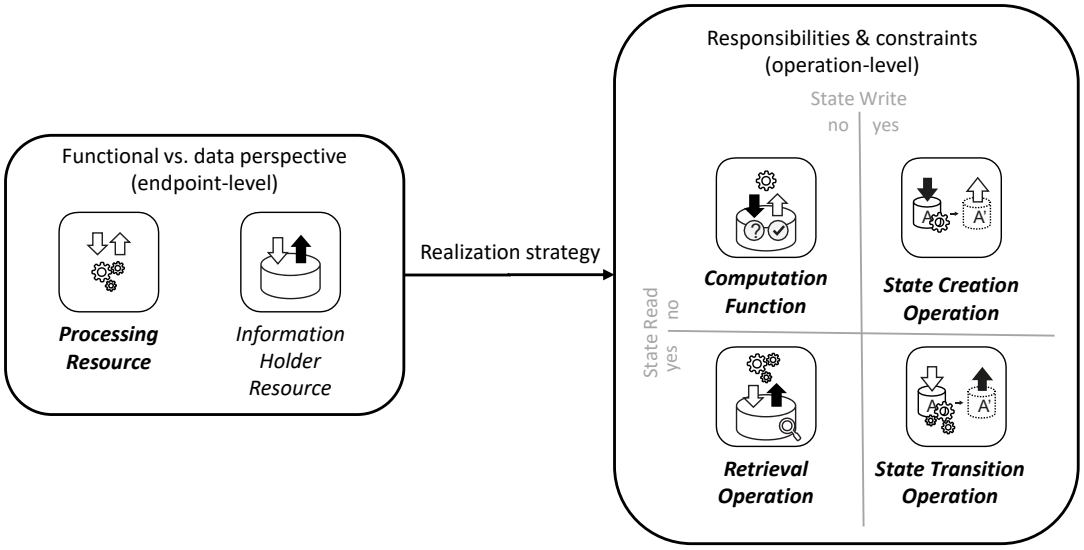


Figure 1: Endpoint role and operation responsibility patterns and their relations (bold pattern names indicate the scope of this paper).

- An information holder "knows and provides information" and a structurer "maintains relationships between objects and information about those relationships"[44].

Specifically, we (re-)use the following role stereotypes: *controller/coordinator* (two roles taken by our *Processing Resource* pattern, this paper) and *information holder* (separate paper).

3 CONTEXT: THE MAP LANGUAGE

3.1 Previously published patterns

Our language is organized into categories, three of which are partially published already [28, 40, 52]:

1. *Foundation patterns*: What type of (sub-)systems and components are integrated? Where should an API be accessible from? How should it be documented?
2. *Responsibility patterns*: Which is the architectural role played by each API endpoint and its operations? How do these roles and the resulting responsibilities impact (micro-)service size and granularity?
3. *Structure patterns*: What is an adequate number of representation elements for request and response messages? How are these elements structured? How can they be grouped and annotated with usage information? [52].
4. *Quality patterns*: How can an API provider achieve a certain level of quality of the offered API, while using its available resources in a cost-effective way? How can the quality trade-offs be communicated and accounted for? [40].
5. *Identification patterns*: How can API endpoints and operations be found in business requirements and domain models? What is the right approach to service decomposition?
6. *Evolution patterns*: How to deal with life cycle management concerns such as support periods and versioning? How to promote backward compatibility and communicate breaking changes? [28]

This paper and its companion cover the responsibility category.

A retrospective of the evolution of the MAP language since 2016 can be found online[5].

3.2 Domain model

We have generalized the concepts and terminology that we found in remote API platforms and integration technologies such as HTTP, gRPC, WSDL/SOAP (to name just a few) into a platform-independent domain model. We described this domain model in a previous EuroPLoP paper [28]; its vocabulary is used throughout our pattern language and also in the following pattern texts.

An *API endpoint* is a provider-side end of a communication channel and a specification of where the *API* resources are located so that *APIs* can be accessed by *API clients*. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints. The *API* exposes *operations*.

3.3 Pattern template

We use the following template for our patterns: The *context* establishes preconditions for pattern applicability. The *problem* specifies a design issue to be resolved. The *forces* explain why the problem is hard to solve – architectural design issues and conflicting quality attributes are often referenced here; a non-solution may be pointed out as well. The *solution* answers the design question from the problem statement, describes how the solution works and which variants (if any) exist. It also gives an example and shares implementation hints. The *consequences* section discusses to which extent the solution resolves the pattern forces; it may also include additional pros and cons and identify alternative solutions. *Known uses* report real-world pattern applications. Finally, relations to other patterns are explained and additional pointers given under *more information*.

[5] https://ozimmer.ch/patterns/2020/04/29/MAPRetrospective.html

4 PROCESSING RESOURCE ROLE AND OPERATION RESPONSIBILITIES

Let us start with an example. The following service contract features the MAP concepts introduced in the previous section:

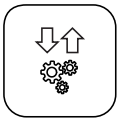
```
data type Customer {"name": D<string>,
                    "address": D<string>,
                    "bday": D<string>}

endpoint type CustomerRelationshipManager
  serves as PROCESSING_RESOURCE
exposes
  operation validateCustomerRecord
    with responsibility COMPUTATION_FUNCTION
    expecting payload "customerRecord": Customer
    delivering payload "isCompleteAndSound": D<bool>
  operation createCustomer
    with responsibility STATE_CREATION_OPERATION
    expecting payload "customerRecord": Customer
    delivering payload "customerId": D<int>
  operation upgradeCustomer
    with responsibility STATE_TRANSITION_OPERATION
    expecting payload "promotionCode": P
    delivering payload P
endpoint type CustomerRepository
  serves as INFORMATION_HOLDER_RESOURCE
exposes
  operation findCustomer
  with responsibility RETRIEVAL_OPERATION
    expecting payload "searchFilter": D<string>
    delivering payload "customerList": Customer*
```

The example also applies the patterns introduced in this paper as decorators (notation: Microservice Domain-Specific Language (MDSL)[6], a new style- and technology-independent service contract modeling language [22]. The above contract specifies a `CustomerRelationshipManager` which is a processing resource, i.e. its primary function is to handle incoming requests. Next, it exposes an operation `validateCustomerRecord` which is a *Computation Function* meaning that it computes a result solely from the input provided in the expected payload. The operation `createCustomer` is different in that it is a write-only *State Creation Operation* that receives notifications about new customers and creates server-side state for a new customer data record accordingly. `upgradeCustomer` has to perform activities causing a server-side state change; thus, it is *State Transition Operation*. The `CustomerRepository` is an *Information Holder Resource* as its primary function is storage and management of customer (master) data. It exposes one *Retrieval Operation* called `findCustomer`.

Table 1 features the pattern names, problem statements, and "bold face" solution summaries.

[6] https://microservice-api-patterns.github.io/MDSL-Specification/



4.1 Pattern: *Processing Resource*

a.k.a. Command Service, Controller Resource, Executor, Processing Endpoint

Context. The functional requirements for an application have been specified, e.g., in the form of agile user stories and/or analysis-level business process models. An analysis of the functional requirements suggests that one or more remote capabilities should be invoked; *Frontend Integration* and/or *Backend Integration* are required and application domain-driven APIs and their clients have to be designed. A (micro-)services architecture and integration infrastructure have been defined initially.

Problem. How can an API provider allow its remote clients to trigger actions in it?

Such actions may be standalone commands (application domain-specific ones or technical utilities) or activities in a business process; they may or may not read and write provider-side application state. A suitable level of abstraction is desired that only exposes the action and hides data as much as possible.

Forces. When invoking provider-side processing upon request from remote clients, general design concerns are:

- Contract expressiveness and service granularity (and their impact on coupling)
- Learnability and manageability
- Semantic interoperability
- Response time
- API security and request/response data privacy
- Compatibility and evolvability

These forces conflict with each other partially. For instance, the more expressive a contract is, the more has to be learned, managed, and tested (w.r.t. interoperability). Finer-grained services might be easier to protect and evolve, but there will be many of them, which have to be integrated [31].

A key decision is whether the endpoint should have activity- or data-oriented semantics. This pattern explains how to emphasize activity; its *Information Holder Resource* sibling [50] focusses on data orientation.

Details. *Contract expressiveness and service granularity.* API designers have to decide how much functionality each API endpoint and its operations should expose. Many simple interactions give the client a lot of control and can make the processing highly efficient, but they also introduce coordination effort and evolution challenges; few rich API capabilities can promote qualities such as consistency but may not suit each client and waste resources. The accuracy of the *API Description* and its implementation also matters. Ambiguities in the invocation semantics harm interoperability and can lead to invalid processing results (which in turn might cause bad decisions to be made and other harm to be caused). Hence, the meaning and side effects of the invoked action (i.e., a self-contained command or part of a conversation) including the representations

Table 1: Problem-solution pairs of the patterns in this paper.

Pattern Name	Problem	Solution
Processing Resource	How can an API provider allow its remote clients to trigger actions in it?	Add a *Processing Resource* endpoint to the API that bundles and wraps application-level activities or commands as its operations (a.k.a. "actions required").
Computation Function	How can a client invoke side-effect-free remote processing on the provider side to have a result be calculated from its input?	Introduce an API operation `f` with `f: in -> out` to an API endpoint (for instance, a *Processing Resource*) that validates the received request message structure, performs the desired function `f`, and returns its result. This *Computation Function* neither accesses nor changes the server-side application state.
State Creation Operation	How can an API provider allow a client to report that something new has happened that is worth capturing (for later processing)?	Add a *State Creation Operation* `f: in -> (out,S')` to an API endpoint (e.g., a *Processing Resource* or an *Information Holder Resource* [50]) that is in essence write-only.
Retrieval Operation	How can information owned or controlled by a remote party (a service provider) be retrieved (to satisfy an information need of an end user or to allow further client-side processing)?	Add a read-access-only operation `f: (in,S) -> out` to an API endpoint to request a report that contains a machine-readable representation of the data that makes up the requested information (this API endpoint may be a *Processing Resource* or an *Information Holder Resource*). Add search, filter, and formatting capabilities to the operation signature (as part of the API contract).
State Transition Operation	How can a client initiate a processing action that causes the server-side application state to advance? How can API clients and API providers share the responsibilities required to perform and control the execution of business processes and their activities?	Introduce an operation in an API endpoint (typically a *Processing Resource*, or an *Information Holder Resource*) that combines client input and current state to trigger a provider-side state change `f: (in,S) -> (out,S')`.

of the exchanged messages must be made clear in the *API Description*, for instance with the help of preconditions, invariants, and postconditions.

Learnability and manageability. An excessive amount of actions leads to hard to understand complexity in the sense that it causes orientation challenges for client programmers, testers, and API maintenance and evolution staff (which might or might not include the original developers); it leads to orientation questions such as how to find and choose the right actions. The more options are available, the more explanations and decision making support have to be given and maintained over time.

Semantic interoperability. Syntactic interoperability is a technical concern for middleware, protocol, and format designers and therefore discussed in our structural representation patterns [52]. That said, the communication parties must also agree on the meaning of the data exchanged before and after any operation is executed; hence, any processors and services must be clear in their *API Description* [28] in what they do and what they do not do (in terms of state changes, idempotency, transactionality, event emission, and downstream resource consumption). Not all of these properties should be exposed publicly to API clients, but still be described.

Response time. Having invoked the remote computation, the client may block until a result becomes available. The longer the client has to wait, the higher the chances that something will break (either on the provider-side or on upstream clients). The network connection between the client and the API may time out sooner or later. An end user waiting for slow results may click refresh, thus putting additional load on an API provider serving the end user application.

API security and request/response data privacy. If a full audit log of all API invocations and resulting server-side processing has to be maintained, e.g., because of data privacy requirements, statelessness on the provider side is an illusion (even if application state is not required from a functional requirement point of view). Personal sensitive information and/or otherwise classified information (e.g., by governments or enterprises) might be contained in the request and response message representations (this also holds true for *Information Holder Resources*). Furthermore, in many scenarios one has to ensure that only authorized clients can invoke certain actions (i.e., commands, conversation parts); for instance, regular employees are usually not permitted to increase their own salary in the employee management systems integrated via *Community APIs* and implemented as microservices. Hence, the security architecture

design has to take the requirements of processing-centric API operations into account – for instance in its Policy Decision Point (PDP) and Policy Enforcement Point (PEP) design and when deciding between Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC). The processing resource is the subject of API security design[7], but also is an opportunity to place PEPs into the overall control flow. The threat model and controls catalogs created by security consultants, risk managers, auditors also must take processing-specific attacks into account, for instance by talking about denial of service (DoS) attacks as well as the probability and impact of the creation of fake orders, fraudulent claims, etc. [21].

Compatibility and evolvability.. The server and the client should agree on the assumptions concerning the input/output representations as well as the semantics of the function to be computed. The client's expectations should match what is offered by the server. The data contract may change over time; if, for instance, units of measure change or optional parameters are introduced, the client must have a chance to notice this and react to it (for instance, by developing an adapter or by evolving itself into a new version, possibly using a new version of the API operation) Ideally, new versions are forward and backward compatible with existing API clients. Our *Evolution Patterns* deal with such concerns in depth [28].

Non-solution. A Shared Database[8] that offers actions and commands in the form of stored procedures[9] could be a valid integration approach (and is used in practice), but creates a single point of failure, does not scale with a growing number of clients, and cannot be (re-)deployed independently [19]. Shared databases with stored procedures do not align well with service design principles such as single responsibility and loose coupling, which is one of the Isolated State, Distribution, Elasticity, Automation and Loose Couping (IDEAL) cloud application properties[10] [10], many of which also apply to on-premises APIs and their implementations.

Solution. Add a *Processing Resource* endpoint to the API that bundles and wraps application-level activities or commands as its operations (a.k.a. "actions required").

How it works. Define one or more operations for the new endpoint that take over a dedicated action responsibility each. *Computation Function*, *State Creation Operation*, and *State Transfer Operation* are common in activity-oriented *Processing Resources*; *Retrieval Operations* should be limited to mere status/state checks here and are more commonly found in data-oriented *Information Holder Resources*. These operation responsibilities might represent a general-purpose or application domain-specific functional system capability (implemented in the API provider or residing in the backend and accessed via an outbound port/adapter) or a technical utility. The request message should make the action explicit and allow the API endpoint to determine which processing logic to execute.

For each of these operations, define a *Command Message* [19] for the request. Add a *Document Message* [19] for the response when required (i.e., when realizing an operation as a *Request-Reply* message exchange [19]). Figure 2 sketches this endpoint-operation design in a UML class diagram.

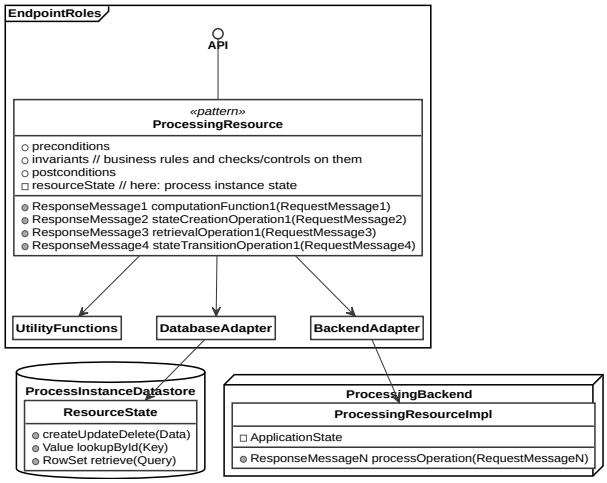


Figure 2: Processing Resources represent activity-oriented API designs. Some operations in the endpoint access and change application state, others do not. Data is only exposed in request and response messages.

Document the endpoint semantics (including pre- and postconditions per operation, as well as invariants) in the *API Description* [28] and/or additional supplemental text. Make the endpoint remotely accessible for one or more API clients by providing a unique logical address.

Decide whether the *Processing Resource* should be a *Stateful Component*[11] or a *Stateless Component*[12]

If invocations of the new API operation cause changes in the (shared) provider-side state, design and decide the approach to consistency management consciously (decisions include strict vs. weak/eventual, optimistic vs. pessimistic locking, etc.). Do not expose transaction management policies in the API (so that they would be visible to the API client), but open and close (or commit, rollback) system transactions inside the API implementation, preferably at the operation boundary. Think about compensating operations a.k.a. sagas for things that cannot be undone easily by system transaction managers. For instance, an email that gets sent in an API call implementation cannot be taken back once it has left the mail server; a second mail "please ignore previous one" has to be sent instead [39], [49].

Example. The Policy Management Backend[13] of the Lakeside Mutual microservices sample contains a stateful *Processing Resource* `InsuranceQuoteRequestProcessingResource` that offers *State Transition Operations* which move an insurance quotation request through various stages. The resource is implemented as an HTTP resource API in Java and Spring Boot. It also contains

[7] https://apisecurity.io/encyclopedia/content/owasp/owasp-api-security-top-10.htm
[8] https://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html
[9] https://en.wikipedia.org/wiki/Stored_procedure
[10] https://www.cloudcomputingpatterns.org/resources/oop17_fehling.pdf
[11] http://www.cloudcomputingpatterns.org/stateful_component/
[12] http://www.cloudcomputingpatterns.org/stateless_component/
[13] https://github.com/Microservice-API-Patterns/LakesideMutual/tree/master/policy-management-backend

`RiskComputationService`, a stateless *Processing Resource* that implements a single *Computation Function* called `computeRiskFactor` (this pattern is introduced Section 4.2) :

```
@RestController
@RequestMapping("/riskfactor")
public class RiskComputationService {
    @ApiOperation(
      value = "Computes the risk factor of a customer.")
    @PostMapping(
        value = "/compute")
    public ResponseEntity<RiskFactorResponseDto>
        computeRiskFactor(
            @ApiParam(
              value = "the request containing relevant
                 customer attributes (e.g., birthday)",
                required = true)
            @Valid @RequestBody
            RiskFactorRequestDto riskFactorRequest) {
        int age =
            getAge(riskFactorRequest.getBirthday());
        String postalCode =
            riskFactorRequest.getPostalCode();
        int riskFactor =
            computeRiskFactor(age, postalCode);
        return ResponseEntity.ok(
            new RiskFactorResponseDto(riskFactor));
    }
```

Implementation hints. Architects and developers that decide to realize *Processing Resources* should take the following advice into consideration:

- Be precise and consistent in the naming of the endpoint (and its operations). For instance, prefer verbs over nouns to express single, action-oriented responsibilities when naming the operations of the endpoint. For operations with domain semantics, avoid generic names such as "execute", "perform", or "do" unless the name of the endpoint (the *Processing Resource*) is self explanatory and expressive enough already. Practice the *Ubiquitous language* [9] of the project; make all names meaningful for domain experts without computer science education and software engineering experience.
- Strive for high cohesion (within a *Processing Resource*) and low coupling (between endpoints). One way of doing so is grouping endpoints by stakeholder/user groups.
- Consider using existing formats such as microformats[14] or ALPS specification(s)[15] to ease client development and promote semantic interoperability .
- Make the resource composable (SOA principle) by avoiding usage of HTTP session state and pushing any state down into a database as soon as possible; in other words, prefer Database Session State[16] over Server Session State[17].
- Try to make the processing of incoming API operation calls idempotent (from a client perspective) to avoid any undesired side effects (realizing the *Idempotent Receiver* pattern [19]). Include this information in the API Description. For instance, HTTP PUT operations should be idempotent, but will still change the provider-side application state. unlike read access, write access is nontrivial to make idempotent (but can be achieved, depending on the operation semantics and message structure design).
- Inform the API user whether invocation results can be cached (as described in the *Conditional Request* pattern [40]).
- Add at least two unit tests per operation to your (automated) test suite, one representing a "sunny day scenario" and another one representing an application domain-level error situation.
- Include the endpoint resource in the DevOps practices for the API and its implementation; for instance, log calls to operations and monitor API performance. Backup the provider-internal application state.
- Version the endpoint adequately, for instance with the help of *Semantic Versioning* and *Version Identifiers* [28]. Strive for backward compatibility when refactoring [46].

Consequences.

Resolution of forces.

+ Business activity- and process-orientation can reduce coupling and promote information hiding.
– In many integration scenarios, activity- and process-orientation would have to be forced into the design, which makes it hard to explain and maintain (among other negative consequences). In such cases, *Information Holder Resource* is a better choice.

The resolution of all other forces is determined on the operation level; see subsequent patterns (Sections 4.2 to 4.5).

Further discussion. A *Processing Resource* can be identified when applying a service identification technique such as *dynamic process analysis* or *event storming* [37]; this has positive effect on the "business alignment" theme in SOA and microservices. One can define one instance of the pattern per backend integration need appearing in a use case or user story; if a single `execute` operation is included in a *Processing Resource* endpoint, it should accept self-describing (or contract-adhering and therefore possible to validate) action request messages and return corresponding, self-contained documents. The request and response messages possibly can be structured according to any of the four structural representation patterns *Atomic Parameter*, *Atomic Parameter List*, *Parameter Tree*, *Parameter Forest* [52].

Other patterns address manageability; see our *Evolution Patterns* [28] for design time advice and remoting patterns books ([7], [43]) for runtime considerations.

Applications of this pattern must make sure not to come across as RPC "tunneled" in a message-based API (and consequently be criticized because RPCs increase coupling, for instance in the time and format autonomy dimensions). Many enterprise applications and information systems do have "business RPC" semantics as they execute a business command or transaction from a user that have to be triggered somehow. According to the original literature and more recent design collections [1], an HTTP resource does not

[14] http://microformats.org/wiki/Main_Page
[15] https://tools.ietf.org/html/draft-amundsen-richardson-foster-alps-01
[16] https://www.martinfowler.com/eaaCatalog/databaseSessionState.html
[17] https://www.martinfowler.com/eaaCatalog/serverSessionState.html

have to model data (or only data), but can represent such business transaction, particularly if it is a long running one.[18]

It is possible to define API endpoints that are both processing- and data-oriented (as many classes in object-oriented programming that combine storage and behavior). Even a mere *Processing Resource* may have to hold state (but wil want to hide its structure from the API clients). Such joint use of *Processing Resource* and *Information Holder Resource* is not recommended for microservices architectures due to the amount of coupling possibly introduced.

Processing Resources require to choose the appropriate conversation or interaction sequence depending on a) how long the processing will take and b) whether the client must receive the result immediately to be able to continue its processing (otherwise, the result can be delivered later). Processing time may be difficult to estimate as it depends on the complexity of the actual function to be executed, the amount of data sent by the client and the load/resource availability of the provider. The request-reply approach at the interface level requires at least two messages that can be exchanged via one network connection, e.g., one HTTP request-response pair in a HTTP resource API. Alternatively, multiple technical connections can be used, for instance, by sending the command via an HTTP POST and polling for the result via GETs in RESTful HTTP.

Decomposing the processing resource to call operations in other API endpoints should be considered (it is rather common that no single existing or to-be-constructed system can satisfy all processing needs, either due to organizational or legacy system constraints). This is where most of the design difficulty lies: How to decompose a *Processing Resource* into the right granularity and set of operations? The "Stepwise Service Design" activity in our emerging Design Practice Repository (DPR)[19] and a future Identification category of our pattern language investigate this problem set.

Known Uses. The Slack Web API[20] is processing-oriented with its `https://slack.com/api/METHOD_FAMILY.method` syntax. It has the notion of "HTTP RPC methods"[21] and even puts a command name in the resource URIs (which is considered a REST anti pattern by some authors in the Web API design community).

The layers of the Domain-Driven Design Sample Application[22], characterized here[23], implement (local) interfaces for *Processing Resources*, `BookingService.java` and `CargoInspectionService.java`.

You can find instances of *Processing Resources* in most integration architectures. Service-Oriented Architectures (SOAs) in enterprises often feature services exposing business capabilities rather than data abstractions in their interfaces; each of these services is a *Processing Resource*. An early example that went into production in early 2003 is the Dynamic Interface of a core banking backend described in an OOPSLA 2004 experience report[24] and in [5].

Terravis provides a process hub for enabling fully digitalized mortgage processes between Swiss land registries, banks, notaries and other parti. It uses a variety of *Command Services* (e.g., Contract Signing) to trigger actions in partners when coordinating inter-organizational business processes [29].

More examples for usage of the pattern in a business information system (a.k.a. enterprise application) context can be found in the telecommunications order management SOA described in [48]. This SOA introduces the notion of business services and application services.

Related Patterns. *Processing Resources* may contain operations hat differ in the way they deal with provider-side state (stateless services vs. stateful processors): *State Transition Operation*, *State Creation Operation*, *Computation Function*, and *Retrieval Operation* (some of which in turn have variants). These specializations also differ w.r.t. client commitment and expectations as expressed in pre- and postconditions and request and response message signatures in the API contract. These four patterns are presented in Sections 4.2 to 4.5 of this paper. The *Information Holder Resource* [50] pattern represents opposite semantics and is an alternative to this pattern.

Processing Resources are commonly exposed in *Community APIs* and *Public APIs*; if this is done, they can be protected with an *API Key* and *Rate Limits* [40]. Their usage is often governed by a *Service Level Agreement* [40] that accompanies the technical API Contract. To avoid that technical parameters creep into the payload in request and response messages, such parameters can be isolated in a *Context Representation* that might come as/be complemented with an *Annotated Parameter Collection*.

The three patterns *Command Message*, *Document Message* and *Request-Reply* [19] are used in combination when realizing this pattern. The *Command* pattern in [13] codifies a processing request and its invocation data as an object and as a message, respectively.[25]

This pattern and its operation-level companions (see above) can be seen as the remote API variant of the general *Command* pattern in [13] and *Application Service* in [2]. Its provider-side implementations often use a *Service Activator* [19].

Other Sources. *Processing Resources* correspond to *interfacers* that provide and protect access to *service providers* in Responsibility-Driven Design (RDD) [44]. Domain-Driven Design (DDD) [9] and this pattern are also related in several ways:

- DDD *Services* are good candidates for remote interface exposure.
- A DDD *Bounded Context* can map and correspond to an API (with several endpoints).
- A DDD *Aggregate* can also map and correspond to an API (with several endpoints, starting with the root entity).
- DDD *Factories* and *Repositories* deal with entity lifecycle management, which involves read and write access to API provider-side application state.
- DDD *Value Objects* can be exposed as Data Transfer Representations (DTRs) in the *Published Language* established by the data part of the API Description (a.k.a. service contract).

[18] Note that HTTP is a synchronous protocol as such; hence, asynchrony has to be added on the application level (or by using QoS headers or HTTP/2). We describe such design in the *Data Transfer Resource* pattern [50] and in [36].
[19] https://github.com/socadk/design-practice-repository
[20] https://api.slack.com/web
[21] https://api.slack.com/methods
[22] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample
[23] http://dddsample.sourceforge.net/characterization.html
[24] http://soadecisions.org/soad.htm#oopsla04

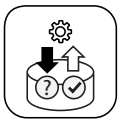
[25] A pattern language for distributed systems design that pulls patterns from other books can be found in [7].

Chapter 6 in "SOA in Practice" [20] is on service classification; it compares several taxonomies including the one from "Enterprise SOA" [24]. Some of the examples in the process services type/category in these SOA books qualify as known uses (these books also include project examples/case studies from domains such as banking and telecommunications).

The online article "Understanding RPC Vs REST For HTTP APIs"[26] talks about RPC vs. REST, but taking a closer look it actually (also) is about deciding between *Processing Resource* and *Information Holder Resource*.

The action resource topic area/category in the API Stylebook[27] provides a (meta) known use for this pattern. Its undo topic[28] is also related.

We now switch from the endpoint level to the operation level; *endpoint roles* may contain any of the following *operation responsibility* patterns.



4.2 Pattern: *Computation Function*

a.k.a. Stateless Computation Operation, Calculation Action, Side-Effect-Free/Stateless Operation

Context. The requirements for an application indicate that something has to be calculated. While the input is available locally and the output will be used in the same place, the calculation should not be run there for cost, efficiency, workload, security, or other reasons.

For instance, an API client might want to ask the API endpoint provider whether some data data meets certain conditions or might want to convert it from one format to another.

Problem. How can a client invoke side-effect-free remote processing on the provider side to have a result calculated from its input?

Forces. The following forces apply when introducing side-effect-free processing on the provider side:

- Networking efficiency vs. data parsimony (message sizes)
- Reproducibility
- Workload management

Details. *Networking efficiency vs. data parsimony.* The smaller messages are, the more messages have to be exchanged to reach a particular goal. Few large messages cause less network traffic, but make the individual request and response messages harder to prepare and process in the communication endpoints.

Reproducibility. Local calls can be logged and replayed rather easily. Outsourcing work to a remote party causes a loss of control and latency, which make it harder to reproduce previous executions.

Workload management. Some computations might require a lot of resources such as CPU time and main memory (RAM); they may run for a long time due to their computational complexity. This may affect the scalability of the provider and its ability to meet the *Service Level Agreement*. If the computation accesses server-side resources (for instance, loggers or backend services), it becomes stateful and cannot be scaled out as easily as stateless computations.

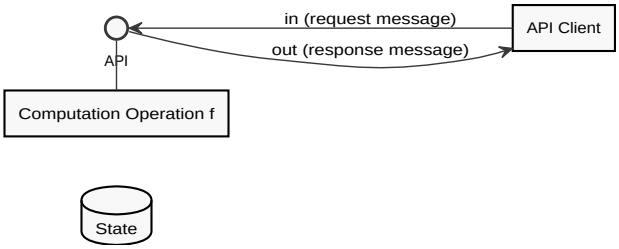


Figure 3: A Computation Function is a stateless operation neither reading nor writing to provider-side storage.

Non-solution. One could perform the required function locally, but this might require to process large amounts of data, which in turn might slow down the client. Eventually such non-distributed approach leads to a monolithic architecture (which has pros and cons).

Solution. Introduce an API operation `f` with `f: in -> out` to an API endpoint (for instance, a *Processing Resource*) that validates the received request message structure, performs the desired function `f`, and returns its result. This *Computation Function* neither accesses nor changes the server-side application state as shown in Figure 3.

How it works. Design request and response message structures that fit the purpose of the function. Consider patterns from the *Structure Category* (for instance, *Parameter Tree* or *Atomic Parameter*) [52] and the *Quality Category* (for instance, *Rate Limit* or *Request Bundle*) [40] when designing the request and response messages that convey the operation responsibility.

Include the function in the *API Description* [28] (in the context of the endpoint it is added to). Define at least one explicit precondition that references the request message (a.k.a. `in` parameters) and one or more postconditions that specify what the response message (a.k.a. `out` parameters) contain (and how this data should be interpreted).

If the computation is resource-intense (CPU, RAM), algorithm and distribution design might have to be rethought to avoid bottlenecks and single points of failure; see conversation pattern *Long-Running Request* [35]. While not directly observable in the functional API contract, this is critical for the API design because it may affect the ability to meet its *Service Level Agreement* (SLA) [40]. CPU and RAM consumptions also impact the components implementing the API; it becomes more challenging to scale the function implementation. Result pre-computation or caching may also come into play. If the API implementation is deployed to a cloud, the cost of renting the cloud service offering also has to be taken into account [40].

There is no need to introduce transaction management here because a mere *Computation Function* is stateless by definition.

[26] https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/
[27] http://apistylebook.com/design/topics/resource-action
[28] http://apistylebook.com/design/topics/undo

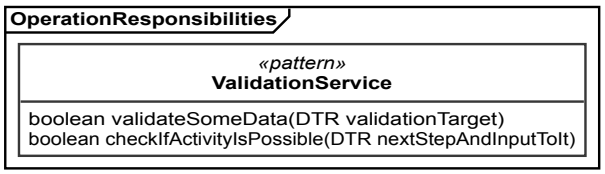


Figure 4: Validation Service variant: arbitrary request data, boolean response (DTR: Data Transfer Representation)

Variant(s). The general, rather simple pattern described above has several variants, *Transformation Service* and *Validation Service* (which both satisfy specialized integration needs) as well as *Long Running Computation* (which is more challenging technically than the general case). Each variant required slightly different request/response message representations.

Transformation Service. A *Transformation Service* implements one or more of the message translation patterns from "Enterprise Integration Patterns" [19] in a network-accessible form. For instance, they might convert from one meta model to another (e.g., customer record schemas used in two different subsystems) or from one meta meta model to another (e.g XML to JSON, JSON to CSV); it does not change the payload/content semantics. Such services typically accept and return *Parameter Trees* of varying complexity.

Validation Service (a.k.a. (Pre-)Condition Checker). To deal with potentially incorrect input, the server should always validate it before processing it and make it explicit in its contract that the input may be rejected. It may be useful for clients to be able to test their input validity explicitly and independently from the invocation of the function for processing it. The API thus breaks down into a pair of two operations:

1. An operation to validate the input without performing the computation.
2. An operation to perform the computation (which may fail due to invalid input unless this input has been validated before).

Step 1 solves the following problem: How can an API provider check the correctness/accuracy of incoming data transfer representations (parameters) and server-side resources (and their state)?

The solution to this problem is to introduce an API operation that receives a *Data Element* and returns an *Atomic Parameter* [52] (e.g., a boolean value or integer) that represents the validation result. The validation primarily pertains to the payload of the request message, but the API implementation may consult the current internal state during the validation (for instance, to look up certain values and calculation rules) as shown in Figure 4.

An example request like "will you be able to process this?" (hat is invoked prior to a call to a *State Transfer Operation*. In case of such "pre-activity-validation", the parameter types can be complex (depending on the activity to be pre-validated); the response might contain suggestions how to correct any errors that were reported. If such "Business Object Validator" includes provider-side application state into the checking process, it morphs into a "Business Rule Validator".

There are many other types of conditions and things to validate, ranging from classifications and categorizations such as `isValidOrder(orderDTR)` and status checks like `isOrderClosed(orderId)` to complex compliance checks, e.g., `has4EyesPrinicipleBeenApplied(...)`. These validations have in common that they return rather simple results (typically, a success indicator and possibly some additional explanations); they are stateless and operate on the received request data exclusively, which makes them easy to scale and move from one deployment node to another.

Long Running Computation. A simple function operation may be sufficient under the following assumptions:

- The input representation is expected to be correct.
- The expected function execution time is short.
- The server has enough CPU processing capacity for the expected peak workload.

However, sometimes the processing will take a noticeable amount of time, and sometimes it cannot be assured that the processing time of a computation will be short enough (for instance, due to unpredictable workload or resource availability on the server or due to varying sizes of input data sent by the client. In such cases, clients should be provided some form of asynchronous non-blocking invocation of a processing function. A more refined design is needed for such *Long Running Computations*, which may receive invalid input and may require to invest a significant amount of CPU time to execute them.

There are different ways of implementing this pattern variant:

- Call over asynchronous messaging. The client sends its input via a request message queue, and the API provider puts the output on a response message queue [19].
- Call followed by callback: the input is sent via a first call, and the result is sent via a callback, which assumes that clients support callbacks [43]
- Long running request (input is posted, a link is provided where the progress can be polled, eventually the result is published at its own link – there is an optional but useful opportunity to use the link to cancel the request and clean up the result when no longer needed) [35].

If we can assume that more than one client is likely to request to perform the same computation over the same input, that the result is deterministic, and that the server has enough storage capacity, then it may be worth it to invest into caching results so that they can be shared across multiple clients.

Examples. A simple, rather self-explanatory example of a *Transformation Service* is shown in Figure 5

A call to find out about the health of a service (a.k.a. heartbeat test message) is another example of a simple command exposed remotely within a *Processing Resource* endpoint (see Figure 6).

Such "I am alive" operations (a.k.a. "ping" calls) accepting test messages) are often added to mission-critical API implementations as part of a systems management strategy (here: fault and performance management). Its pre- and postconditions are simple; its API contract is sketched in the above UML snippet. Neither system

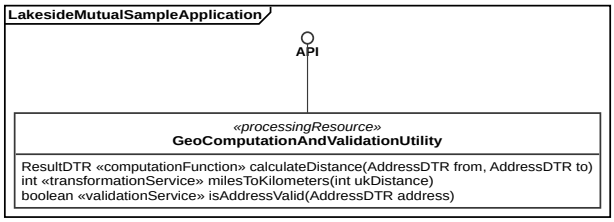


Figure 5: A Processing Resource providing Transformation Services

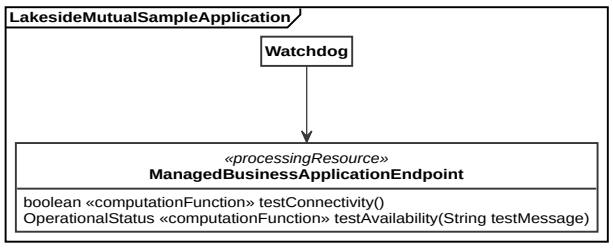


Figure 6: Examples of Validation Services: Health Check operations

transactions nor business-level compensation (undo) is required in this simple example.

Implementation hints. Architects and developers that decide to realize stateless *Computation Operations* should take the following advice into consideration:

- Name the operation in such a way that the operation responsibility and the pattern variant (*Long Running Computation*, *Validation Service*, *Transformation Service*) get clear intuitively. For instance `hasXYZProperty` or `meetsConditionABC` are good names for *Validation Services* (if `XYZ` and `ABC` are domain terms), while `check` or `test` are rather generic and therefore less expressive and intuitive. They do not unveil the validation goal.
- Offer one and only one way to express the input and minimize the number of options for the presentation of the output.
- Specify the postcondition formally or semi-formally; consider the same for the precondition.
- Respect the variants of *Computation Function* (*Long Running Computation*, *Validation Service*, *Transformation Service*) by adding suited test cases.
- Maintain computation/validation services and computation input/validated data together (for instance, use the same version control system and repository).
- When realizing *Transformation Services* and *Validation Services*, consider to integrate an off-the-shelf automation of syntactic validations such as JSON or XML schema validations, as validation logic takes effort to write and maintain (and typically this effort is not contained in project budgets). You can either include an adequately licensed, mature enough library or call an external service.
- Being stateless, *Computation Operations* can be realized as HTTP GETs (unless its `in` parameters are too complex to be represented as path and/or query parameters). When using HTTP to invoke long-running *Computation Operations* (or those returning multiple complex results), the Multipart Content-Type[29] can be used to bundle several responses.
- Keep track of the performance overhead of frequent calls to externalized *Computation Operations*; in case of problems, consider the *Request Bundle* pattern, scale out, or revert to local calls.

Consequences.

Resolution of forces.

+ Workload management is simplified because stateless operations can be moved freely.
– Reproducability and auditability suffer because an external dependency is introduced that cannot be controlled by the client; it has to trust the provider that multiple subsequent calls are possible and will return the same result.
– Message size might increase because stateless servers cannot retrieve any intermediate results from their own data stores.

Further discussion. Exposing API calls with business domain semantics such as computations, transformations and data validation is a key principle in SOA and tenet in microservices implementations. The pattern can help resolve the forces and contribute to application health if implemented properly. If exposing a transformation or validation operation as a remote service is too costly, a local library-based API is a cheaper alternative.

From a security point of view, the request message of a validation or transformation often has low to medium needs, but has to avoid denial-of-service attacks; the response message often has lower protection needs (if it is less expressive).

By definition, implementations of the pattern do not change application state on the server (possibly except for access logs and temporary or permanent storage of validation results, if/as needed to meet security requirements such as non-repudiation). They are therefore easy to scale and to move, which makes them eligible to cloud deployments.

Known Uses. Serverless computing lambdas, deployed to public clouds such as AWS or Azure, may be seen as *Computation Functions* as well (unless they are combined with cloud storage offerings, which makes them stateful).

Online JSON and JSON schema validators operate purely input-based and qualify as *Validation Services*; they do not have to take server-internal state into account (other than the schema definition and their metametamodels). The same holds for image manipulation and PDF processing tools, both of which are abundant on the Web today (*Transformation Service*).

A basic known use of the *Validation Service* variant of this pattern can be found in the `Cargo` root entity and the `RouteSpecification` in the Cargo Aggregrate[30] of the Domain-Driven Design Sample

[29] https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html
[30] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/domain/model/cargo

Application[31]. It implements several validation routines (that are not exposed in the remote API of the application).

A major German car manufacturer offers a REST level 2 user profile management microservice for all its clients; the *analysis* endpoint of this service qualifies as a known use of the *Validation Service* variant: It offers a `POST` operation to validate data strictly, suggests corrections, and converts addresses and phone numbers to country-specific standards (without storing them); profile updates are also supported by the microservice, but at another endpoint address. Swagger/Open API contracts are exposed in an *API Description* portal/website [28].

Terravis, a process integration platform [4], offers an API to internal clients to calculate a valid payment date. The operation is called by specifying a desired payment date. Depending on certain rules (e.g., taking bank holidays into accounts as well as weekends), the next feasible payment date is calculated and returned to the caller. This operation does not change the provider-side application state, all required input comes from the received request message. Moreover, Terravis offers *Solution-Internal APIs* for generating various business documents such as contracts. Data is passed from the calling clients to the document generation API, which validates the data for completeness and conformance with certain business rules and then generates a PDF document statelessly.

Related Patterns. The pattern compares to its siblings as this:

- Just like a *Retrieval Operation*, a *Computation Function* does not change the application state (but delivers nontrivial data to the client); it receives all required data from the client, whereas a *Retrieval Operation* consults provider-side application state (in read-only mode).
- Both *State Creation Operations* instances and *Computation Functions* receive all required data from the client; a *State Creation Operation* changes the server-side application state (write access) whereas a *Computation Function* preserves it (no access). The client of a *Computation Function* usually has a higher expectation w.r.t. the response than a client that invokes an *State Creation Operation* to report an event (and only requires a positive confirmation).
- A *State Transition Operation* also returns non-trivial data (like *Retrieval Operation* and *Computation Function*, but it also changes the server-side application state. Input comes from the client but also from the server-side application state (read-write access).

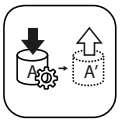
The *Service* pattern in Domain-Driven Design (DDD) includes similar semantics (but is broader) and can help to identify *Computation Function* candidates during endpoint identification [42].

Other Sources. Service types are a topic covered by SOA literature from the early 2000s, e.g., "Enterprise SOA" [24] and "SOA in Practice" [20]. While the service type taxonomies in these books are more focussed on the overall architecture, some of the basic services and utility services have responsibilities that do not require read or write access to provider/server state and therefore qualify as instance of this pattern an its variants.

[31] http://dddsample.sourceforge.net/characterization.html

The design-by-contract approach in the object-oriented programming method and language Eiffel [30] includes validation into business commands/domain methods and automates pre- and postcondition checking. This program-internal approach can be seen as an alternative to external *Validation Services* (but also as a rather advanced known use of it).

A lot of online resources on serverless computing exist. One starting point is the web site and blog "Serverless"[32] by J. Daly.



4.3 Pattern: *State Creation Operation*

a.k.a. Write-Only Operation, Data Insertion Operation

Context. An API endpoint has been introduced. The API client has expressed its API wants and needs, for instance in the form of user stories and/or given-when-then clauses[33]; non-functional requirements have been elicited as well.

The API client(s) would like to inform the API provider about new client-side incidents without requesting any further server-side processing that would require any immediate response to be returned beyond a simple "got it" acknowledgment (and, provider-internally, initialization of application state).

For instance, the client might want to kick off a long-running business transaction (like an order management and fulfillment process) in the provider or report the completion of a client-side batch job (like the bulk re-initialization of a product catalog). Such creation events cause data to be inserted on the provider side, but this does not become visible to the client.

Problem. How can an API provider allow a client to report that something new has happened that is worth capturing for later processing?

Forces. The following forces have to be taken into account:

- Coupling tradeoffs (accuracy and expressiveness vs. information parsimony)
- Consistency effects
- Timing considerations
- Reliability considerations

Details. *Coupling tradeoffs (accuracy and expressiveness vs. information parsimony).* To ease processing on the provider side, the incoming report should be self-contained so that it is independent from other events. To streamline report construction on the client side, save transport capacities and hide implementation details, it should only contain the bare minimum of information the API provider is interested in.

Consistency effects. If the providers-state can or should not be read, it becomes more difficult to validate that the provider-side processing caused by incoming requests does not break and invariants and other consistency properties.

Timing considerations. The client-side occurrence of an incident may differ from the moment it is reported and the time when the incident report finally reaches the provider. It may not be possible

[32] https://www.jeremydaly.com/serverless/

[33] https://www.martinfowler.com/bliki/GivenWhenThen.html

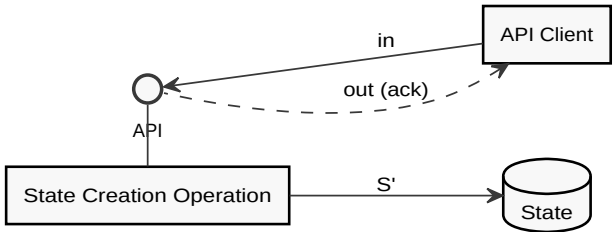


Figure 7: A State Creation Operation has the responsibility to write to provider-side storage, but cannot read from it.

to determine the sequencing/serialization of incidents happening on different clients (time synchronization is a general theoretical limitation and challenge in any distributed system; logical clocks have been invented for that reason).

Reliability considerations. Sometime reports cannot be processed in the same order in which they were produced and sent. Sometimes reports get lost or the same reports is transmitted and received multiple times. It would be nice to acknowledge that the report causing state to be created has been processed properly.

Non-solution. One could simply add yet another API operation to an endpoint without any special semantics (and pattern usage). If this is done, the specific integration needs and concerns described above have to be made explicit in the API documentation and usage examples; there is a risk of making implicit assumptions that get forgotten over time. Ths can cause undesired extra efforts for client developers and API maintainers when they find out that their assumptions no longer hold. Furthermore, Cohesion within the endpoint might be harmed; DevOps staff has to guess where and how to deploy the endpoint implementation (for instance, in certain cloud environments and container managers). Load balancing becomes more complicated.

Solution. Add a *State Creation Operation* `f: in -> (out,S')` to an API endpoint (e.g., a *Processing Resource* or an *Information Holder Resource*) that is in essence write-only. Here 'in essence' means that such operations might have to read some state, e.g., to check for duplicate keys in existing data before creation, but their main purpose should be state creation.

The design is sketched in Figure 7.

How it works. Let such *State Creation Operation* represent a single business incident that does not mandate any instant reaction from the provider-side endpoint; it is free to simply store the data, acknowledge it, or perform further backend processing. Let the client receive a mere "got it" acknowledgment or identifer (for instance to be able to enquire about the state and resend the operation in case of transmission problems).

Describe the abstract and the concrete syntax as well as the semantics of the incident report (i.e., the incoming state creation messages) and the acknowledging response (if any) in the *API Description* [28]. Express the operation behavior in (rather simple) pre- and postconditions.

State Creation Operations may or may not have fire-and-forget semantics. In the latter case, give each state item caused by calls to instances of this pattern a unique id (for duplicate detection and removal). Include a timestamp to capture the time when the reported incident happened (according to the client-side clock).

Unless you write to an append-only event store, perform the required write/insert operation in its own system transaction whose boundaries match that of the API call (but are not visible to the API client). Let the processing of the *State Creation Operation* appear to be idempotent.

The request messages accepted by a *State Creation Operation* contain all data that is required to describe the incident that has happened (but not more) in its request message, often in the form of a *Parameter Tree* [52], possibly annotated with *Metadata Element*. Past tense is often used to name events (e.g., "customer entity created"). The response message typically only contains a basic and simple "report received" element, for instance, an *Atomic Parameter* [52] containing an explicit positive acknowledgment or an *Atomic Parameter List* [52] combining an error code with an error message (forming an *Error Report*).

Variants. A prominent variant of this pattern is *Event Notification Operation*, notifying the endpoint about an external event without assuming any visible provider-side activity and thus realizing event sourcing[34]. Such *Event Notification Operations* can report that data has been created, updated (fully or partially), or deleted elsewhere. Unlike in most implementations of stateful server-side processing, the incoming event is only stored as-is, but the application state is not updated instantly. If the most recent state is required later on, all stored events (or all events up to a certain point in time when snapshots are taken) are rather replayed and the application state is calculated. This makes the event reporting fast, at the expense of slowing down the later state lookup. An additional benefit of event sourcing is that time-based queries can be performed as the entire data manipulation history is available in the event journal. Modern event-based systems such as Apache Kafka support such replays in their event journals and distributed transaction logs.

Event Notification Operations and event sourcing can form the base of Event-Driven Architectures (EDAs); see related pattern languages for advice [38].

Another variant of this pattern is a *Bulk Report*: The client reports multiple related events (in a single call in the form of a *Request Bundle* or in separate ones) that all pertain to the same or to different entities (domain objects), for instance to simplify processing of each individual one or to create a full audit log.

Examples. In the online shopping scenario, message such as "new product XYZ created" send from a product management system or "customer has checked out order 123" from an online shop qualify as examples.

Figure 8 gives an example in a fictitious insurance company.

Steps 5 and 6 of the demo "Domain-Driven Service Design with Context Mapper and MDSL"[35] feature an instance of this pattern and introduces the MDSL decorator for it:

```
"STATE_CREATION_OPERATION" @PaperItemDTO
createPaperItem (String who, String what, String where);
```

[34] https://martinfowler.com/eaaDev/EventSourcing.html
[35] https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html

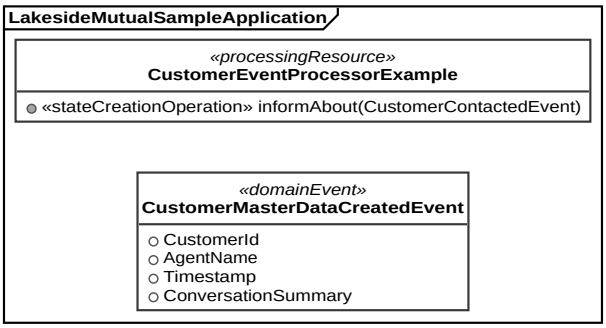


Figure 8: Example of a State Creation Operation: Event Processor

Implementation hints. Architects and developers that decide to apply and realize a *State Creation Operation* should take the following advice into consideration:

- Apply a recognized analysis and design practice to identify *State Creation Operations* and their endpoints, for instance event storming[36].
- Test with representative data, include different timing conditions and error situations in the test plan.
- Acknowledge the recommended practices for event sourcing, for instance to avoid unnecessary/undesired data coupling. Small event messages might have to cross-reference each other; large messages might be self-contained but take longer to process. Consider usage of Command-Query-Responsibility-Segregation (CQRS)[37] to be able to design and deploy the write and the read part of an API separately (for instance, for performance reasons when facing high workloads). A tutorial on event sourcing and CQRS in the context of Domain-Driven Design can be found on the Context Mapper website[38].
- Make sure to use universal data formats, especially for the time stamps of incoming messages. An example of such format is the CloudEvents specification[39].
- When implementing the pattern in HTTP resource APIs, use the POST method by default, but also consider PUT. Consult [1] for further advice regarding when to prefer which verb.

Consequences.

Resolution of forces.

+ Loose coupling is promoted because client and provider do not share any application state, the client merely informs the provider about activities on its side. No provider-side state is read when a request message arrives.

– No provider-side checks are possible due to missing state reads. Hence, consistency cannot be ensured fully when write-only operations are used.
– Time management remains a difficult design task for the same reason.
– Reliability might suffer if no acknowledgment or state identifier is returned; if it is returned, the API client has to make sure to interpret it correctly (for instance, to avoid unnecessary or premature resending of messages).

Further discussion. Exposing write-only API operations with business semantics that report external events is a key principle of EDAs; we discussed it in the *Event Notification Operation* variant. In replication scenarios, events represent state changes that have to be propagated. In Domain-Driven Design, domain event[40] sourcing is the recommended practice to integrate Aggregates (both within the same and in different Bounded Contexts) because it decouples them and allows replaying events up to the current state in case of failures that lead to consistency issues [42].

The pattern is easier to apply, implement and test than its more comprehensive *State Transition Operation* peer, but leaves room for interpretation on the receiver side (here: API provider):

- What should be done with arriving reports, should they be simply stored locally, processed further, or passed on? Does provider-side state have to be accessed minimally, for instance to check the uniqueness of keys?
- Does the report processing change the behavior of future calls to other operations in the same endpoint?
- Are the operations idempotent? How to ensure strict or eventual consistency (events can get lost due to the fallacies of distributed computing, and there is a tradeoff between consistency and availability according to the CAP and BAC theorems [34]).

State Creation Operations are sometimes exposed in *Public APIs*; if this is done, they can be protected with an *API Key* and *Rate Limits* [40]. Introducing a *Rate Limit* can be rather critical, as events may happen with high throughput, depending on the domain/source.

This pattern covers scenarios in which an API client notifies a known API provider about an incident. An API provider notifying its clients via callbacks and publish-subscribe mechanisms is another approach covered in other pattern languages and middleware/distributed systems books [15], [19], [43].

Known Uses. Known uses of this pattern are common in enterprise information systems and public Web APIs:

- The `submitReport` operation of the Handling Report Service[41] in the Domain-Driven Design Sample Application[42] implementing a cargo tracking scenario allows external clients to notify the application about handling events such as a container arriving at a particular port.
- The Slack Event API[43] is a comprehensive implementation of API-based event processing.
- The article "Know the Flow! Microservices and Event Choreographies"[44] by B. Rücker introduces *event command transformations* and implements an example that includes a usage

[36] https://contextmapper.org/docs/event-storming/
[37] https://martinfowler.com/bliki/CQRS.html
[38] https://contextmapper.org/docs/event-sourcing-and-cqrs-modeling/
[39] https://cloudevents.io/
[40] https://martinfowler.com/eaaDev/DomainEvent.html
[41] https://github.com/citerus/dddsample-core/blob/master/src/main/java/com/aggregator/HandlingReportService.wsdl
[42] http://dddsample.sourceforge.net/characterization.html
[43] https://api.slack.com/events-api
[44] https://www.infoq.com/articles/microservice-event-choreographies

of this pattern ("order placed") as well as the sibling pattern *State Transition Operation*. The articles makes the case for workflow engine usage on the microservice API provider side.

The pattern is often applied in enterprise settings as well (showing how common us of the pattern is):

- Banks can request a new process at the platform Terravis. By calling a "start process" operation, a new process is instantiated and state for it is managed separately from other process instances from now on. Thus, this start operation is a *State Creation Operation*.
- A provider of a Swiss banking solution software implemented a microservice architecture based on events that are implemented as *State Creation Operations*. For example, such notifications report modifications of business objects to interested microservices, which in turn can update their local data or initiate processes, e.g., checking a customer against a blacklist.

Related Patterns. The endpoint-level role patterns *Processing Resource* and *Information Holder Resource* [50] may contain instances of this pattern; its sibling patterns (other operation responsibilities) are *State Transition Operation* (and its variants), *Computation Function*, and *Retrieval Operation* (and its variants). A *State Transition Operation* usually identifies a provider-side state element in its request message (for instance, an order id or serial number of a staff member); *State Creation Operations* do not have to do this (but might).

Event-Driven Consumer and *Service Activator* in [19] describe how to trigger message consumption asynchronously. Chapter 10 in "Process-Driven SOA" features patterns for integrating events into process-driven SOAs [16].

In Domain-Driven Design (DDD), the *Domain Event* pattern [42] has similar semantics and can help to identify *State Creation Operations* in the variant *Event Notification Operation* during endpoint identification.

Other Sources. Instances of this pattern may participate in long running and therefore stateful *conversations* [17].

CQRS[45] and event sourcing[46] are described by M. Fowler and other authors. As event sourcing and domain events in DDD have gained momentum and popularity in recent years, one can find a lot of best practice advice for modeling and implementing them, for instance in presentations and articles by V. Vernon[47], and M. Plöd[48], and C. Richardson[49]. Other online resources on event sourcing and CQRS can be found at InfoQ[50] and DZone[51]. The Context Mapper DSL and tools support DDD modeling, model refactoring as well as diagram and service contract generation.

[45] https://martinfowler.com/bliki/CQRS.html
[46] https://martinfowler.com/eaaDev/EventSourcing.html
[47] https://vaughnvernon.co/
[48] https://de.slideshare.net/mploed/presentations
[49] https://microservices.io/patterns/data/event-sourcing.html
[50] https://www.infoq.com/eventsourcing/
[51] https://dzone.com

The open source Software/Service/API Design Practice Repository (DPR)[52] features a seven-step service design method to carve out API endpoints and their operations.

4.4 Pattern: *Retrieval Operation*

a.k.a. Read-Only Operation, State Lookup Operation, Query, Data Extractor

Context. An API endpoint has been established; functional and non-functional requirements for it have been specified. However, the operations of these resources do not cover all required integration capabilities yet; the API consumer(s) also demand read only access to large amounts of structured, possibly aggregated data. This data can be expected to be structured differently than in the underlying domain model; for instance, it might pertain to a particular time interval or subdomain element (like a product category or customer profile group). The information need arises either ad hoc or regularly, e.g., at the end of/for a certain time interval (such as week, month, quarter, or year).

Problem. How can information owned or controlled by a remote party (a service provider) be retrieved (to satisfy an information need of an end user or to allow further client-side processing)?

Related sub-problems are:

- How can data model differences be overcome and data be aggregated and combined with information from other sources?
- How can clients influence the scope and the selection criteria for the retrieval results?
- How can the time frame for reports be specified?[53]

Forces. The following top-level forces have to be resolved when exposing data in API operations:

- Veracity, variety, velocity, volume (the four Vs in big data).
- Workload management
- Networking efficiency vs. data parsimony (message sizes)

Details. *Veracity, variety, velocity, and volume.* Data comes in many forms and client interest in it varies. See this infographic[54] for an introduction to and illustration of the challenges when exposing and analyzing large amounts of data.

Workload management. See explanation of this force in *Computation Function* pattern (Section 4.2 of this paper).

Networking efficiency vs. data parsimony (message sizes). Also see explanation in *Computation Function* pattern (Section 4.2).

Non-solution. It is hard to imagine a distributed system that does not require some kind of retrieval and query capability. One could replicate all data to its users "behind the scenes" periodically, but such approach has major deficiencies w.r.t. consistency, manageability, and data freshness.

[52] https://github.com/socadk/design-practice-repository
[53] Note that the scheduling of periodic query execution (a form of batch processing) is out of scope here.
[54] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

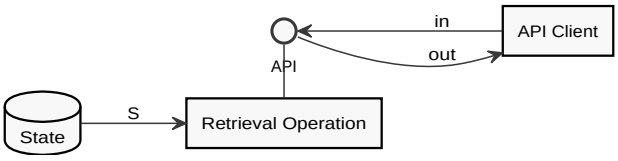


Figure 9: A Retrieval Operation reads from, but does not write to provider-side storage. Searching and filtering may be supported.

Solution. Add a read-only operation `f: (in,S) -> out` to an API endpoint to request a report that contains a machine-readable representation of the requested information (this API endpoint may be a *Processing Resource* or an *Information Holder Resource*). Add search, filter, and formatting capabilities to the operation signature.

How it works. Access the provider-side state in read-only mode. Make sure that the pattern implementation does not change application/session state on server (except for access logs etc.) as shown in Figure 9. Document this behavior in the *API Description*.

For simple retrievals, one can use an *Atomic Parameter List* to define the query parameters for the report and return the report as a *Parameter Tree* or *Parameter Forest* [52]. In more complex scenarios, a more expressive query language (such as GraphQL[55] with its hierarchical call resolvers or SPARQL[56], used for big data lakes) can be introduced; the query then describes the desired output declaratively (i.e., as an expression formulated in the query language); it can travel as an *Atomic Parameter* [52] string. Such expressive, highly declarative approach supports the "variety" V (one of the four Vs introduced above).

Adding support for *Pagination* [52] is common and advised if result collections are large (the "volume" V of the four big data Vs). The output can be streamlined when the request contains a *Wish List* or *Wish Template* [40].

Supporting data access settings (i.e., transaction boundary and isolation level) may be required in the operation implementation.

Examples. In an online shopping example, an analytic *Retrieval Operation* "show all orders customer ABC has placed in the last 12 months".

In the Lakeside Mutual example, we can define two operations to find customers as illustrated in Figure 10. CRM stands for Customer Relationship Management; the `allData` parameter is a simple *Wish List*, allowing to return either an *Embedded Entity* or a *Linked Information Holder*.

A code-level example is:

```
curl http://localhost:8080/claims?limit=10\&offset=0
```

```
@GET
public ClaimsDTO listClaims(
  @DefaultValue("3")
  @QueryParam("limit") Integer limit,
  @DefaultValue("0")
  @QueryParam("offset") Integer offset,
  @QueryParam("orderBy") String orderBy
```

[55] https://graphql.org/

[56] https://en.wikipedia.org/wiki/SPARQL

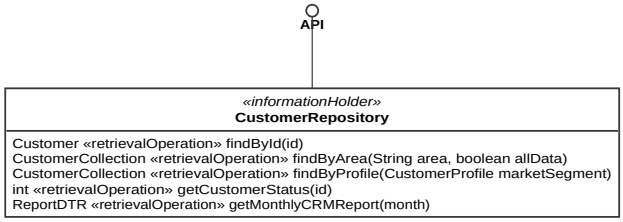


Figure 10: Examples of Retrieval Operations: search, filter, direct access

```
) {
  List<ClaimDTO> result = [...]
  return new ClaimsDTO(limit, offset,
    claims.getSize(), orderBy, result);
}
```

Steps 5 and 6 of the demo "Domain-Driven Service Design with Context Mapper and MDSL"[57] features a an instance of this pattern and introduces the MDSL decorator for it:

```
"RETRIEVAL_OPERATION" Set<@PaperItemDTO>
lookupPapersFromAuthor (String who);
```

Variants. Several variants of this pattern exist, for instance *Status Check* a.k.a. Progress Inquiry/Polling, *Time-Bound Report*, and *Business Rule Validator*.

A *Status Check* has rather simple in and out parameters (e.g., two *Atomic Parameter* instances): an id is passed in and a status code (int) or state name (from an enumeration) are returned.

A *Time-Bound Report* typically specifies the time interval(s) as an additional query parameter (or set of parameters); its responses then contain one *Parameter Tree* per interval.

A *Business Rule Validator* is similar to the *Validation Service* variant of a *Computation Function*. However, it does not validate data that is passed on, but retrieves this data from the provider-side application state. A list of identifiers of entities already present in the server (validation target) might be included in the request. One example of a hybrid *Business Rule Validator* is a check whether the provider will be able to process this business object in the current state the conversation wit the client. Such validator can be invoked prior to a call to a *State Transfer Operation* that primarily works on the business object that is passed in, but also includes provider-side application state into the checking process. In an online shopping example, "check whether all order items point to existing product that are currently in stock" is an example of such validator.

Implementation hints. Architects and developers that decide to realize *Retrieval Operations* should take the following advice into consideration:

- Realize *Retrieval Operations* as HTTP GETs when designing HTTP resource APIs, possibly cached, if the input is simple enough to be expressed in path and query parameters; use POST otherwise.
- Make the types of the query parameters explicit, for instance by introducing *Metadata Elements*.

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

- Provide one example per query capability that includes sample responses and rationale for their selection; the more powerful and expressive the descriptive/declarative a query language is, the harder it is to create and maintain query statements/expressions that do exactly what you want — and continue to do so in the long run.[58]
- Invest in data quality and related metrics/management procedures to achieve the required veracity (one of the four Vs from the forces section).
- Consider introducing *Rate Limits* and *Conditional Requests* to reduce server-side workload.
- Keep track of the workload caused by *Retrieval Operation* calls.
- Consider caching if performance becomes a concern, but be advised that caches are not easy to design, test, and maintain (cache invalidation, for instance, is one of two hard problems[59] in computer science).
- Queries (reads) can be separated from commands (writes) to optimize the respective performance of these two different channels; each channel then has its own API. This architectural pattern is called Command-Query Segregation/Separation (CQRS)[60].
- Acknowledge the concepts, methods, tools and related best practices from the business intelligence and data warehouse community and the big data movement (e.g., use Extract-Transform-Load (ETL) staging to achieve separation of operational and analytical data processing).

Consequences.

Resolution of forces.

+ Workload management: Due to their read-only nature, *Retrieval Operations* can scale by replicating data.
+ Networking efficiency vs. data parsimony (message sizes): *Retrieval Operations* can make full use of identifiers, can fetch, cache, and optimize local data on demand (note: there is no need for all of this data to appear in the request).
– May become a performance bottleneck if user information needs and query capabilities do not match.

To resolve 4-V force (veracity, variety, velocity, and volume) additional patterns, technology-level practices and design tactics are required; we highlighted some in the solution description above.

Further discussion. Usage of *Pagination* is common to address "the volume V"; the "velocity V" can not be easily supported with standard request-reply retrievals; the introduction of stream processing (which is out of our scope here) can be considered instead.

If query responses are not self explanatory, metadata can be introduced to reduce risk of misinterpretations on consumer side.

From a security point of view, the in message often has low to medium data protection needs; however, the request message may contain secure credentials to authorize access to sensitive information and has to avoid denial-of-service attacks. The out message protection requirements might be more advanced, as the report might contain business performance data or sensitive personal information such as health case records. OWASP has published an API Security Top 10[61] that any API should respect, especially those dealing with sensitive and/or classified data.

Retrieval Operation instances are commonly exposed in *Public APIs*; if this is done, they can/should be protected with an *API Key* and *Rate Limits* [40]. The *Rate Limits* for queries might restrict the number of queries and/or the number of results; GitHub and Google Search supply usage examples for these patterns.

Time-Bound Report services can use denormalized data replicas and apply the extract-transform-load staging commonly used in data warehouses. Such services are common in *Community APIs* and *Solution-Internal APIs*.

Known Uses. Known uses of this pattern are very common in enterprise information systems and public Web APIs:

- eBay has a traffic report API operation[62]
- The `Cargo Repository` in the Cargo Aggregate[63] of the Domain-Driven Design Sample Application[64] implements two basic find operations `Cargo find(TrackingId trackingId)` and `List<Cargo> findAll()`.
- The Open Weather Map API[65] illustrates this pattern in its many lookup options (expressed as parameters). For instance, historic weather data per city is provided by History Bulk[66]. Note that only the responses are batched/bulked in a file; the requests are single/individual ones.
- The Slack Web API has a `files.list` method[67]. While not truly RESTful (but HTTP-based), this call shows a typical parameter and query response message structure. Another known use can be found in the Force.com API[68]. It works with a *Salesforce Object Query Language (SOQL)* to define the query parameters.
- The Swiss eGov initiative offers a central company lookup WSDL/SOAP service called *Zefix*[69] with operations such as `searchByName`.

GraphQL queries (but not its mutations) and restSQL[70] can be seen as advanced, particularly flexible instances of the pattern (designed to avoid over-fetching).

The pattern is often applied in enterprise settings as well (just to show how common the pattern is:

- Some of the 1000+ services in [5] qualify as known uses of this pattern, for instance, "Kundengesamtübersicht" (i.e., a wild carded search for customers and an overview of their activities and products used). See this article[71] for a business-oriented architecture overview.

[58] have you ever have to debug a complex XPath/XQuery expression that you had not touched in a long time?
[59] https://martinfowler.com/bliki/TwoHardThings.html
[60] https://martinfowler.com/bliki/CQRS.html
[61] https://owasp.org/www-project-api-security/
[62] https://developer.ebay.com/api-docs/sell/analytics/resources/traffic_report/methods/getTrafficReport
[63] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/domain/model/cargo
[64] http://dddsample.sourceforge.net/characterization.html
[65] http://openweathermap.org/api
[66] http://openweathermap.org/history-bulk
[67] https://api.slack.com/methods/files.list
[68] https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/dome_query.htm
[69] https://www.e-service.admin.ch/wiki/display/openegovdoc/Zefix+Webservice
[70] http://restsql.org/doc/Overview.html
[71] https://subs.emis.de/LNI/Proceedings/Proceedings175/378.pdf

- The same holds for a subset of the business and application services in the order management SOA presented in [48]. An example is the customer master data lookup performed during phone number and address validation.
- A large Swiss banking software provider provides its clients an API to fetch aggregated data in the context of domain-specific use cases, e.g., a 360-degree view on a customer.

Terravis' parcel information retrieval API [4] provides examples of more sophisticated *Retrieval Operations* that do not directly translate to database queries. Because land registry data in Switzerland is federated, the parcel information retrieval API is a facade that hides the location of the data from the client. Internally, a request is resolved and routed to the responsible land registry, which in turn offers standardized APIs for land register data retrieval. The main operation retrieves a list of parcels by an electronic parcel ID (eGRID, German: "Elektronische Grundstücks-ID"). IDs can be resolved by querying by municipality, owner names, and so on. Requests have size limitations, i.e., only ten parcels may be queried at once in order to manage load on land registry systems. Operations may return historical data if this is desired. Depending on the number of managed objects, queries can be paginated (as described in our *Pagination* pattern [52]) or the amount of data can be set to different levels (e.g., full, partial, full history) with a *Wishlist* [40].

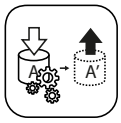
Related Patterns. The endpoint pattern *Processing Resource* and all types of *Information Holder Resources* [50] may expose *Retrieval Operations*. The *Pagination* pattern [52] is often applied in *Retrieval Operations*.

The sibling patterns are *State Transition Operation*, *State Creation Operation*, and *Computation Function*. An *State Creation Operation* pushes data from the client to the API provider, whereas a *Retrieval Operation* pulls data; both *Computation Function* and *State Transition Operation* can support unidirectional data flows and bidirectional ones.[72]

Other Sources. The is a large body of literature on database design and information integration, including data warehouses [23]. Chapter 8 in the *RESTful Web Services Cookbook* by [1] discusses queries (in the context of HTTP APIs).

"Implementing Domain-Driven Design" [42] talks about Query Models in Chapter 4 (in the section on CQRS). Endpoints that only expose *Retrieval Operations* form the Query Model in CQRS.

4.5 Pattern: *State Transition Operation*



a.k.a. Read-Write Operation, Data Change Operation, Business Activity Processor

Context. It has been decided to expose business functionality in an API. The functionality should be decomposed into multiple activities, whose execution state should be visible in the API so that clients can advance it. For example, functionality that is part of longer-running business processes might require data exchanges including incremental updates and coordinated application state management to move process instances from initiation to termination in a stepwise fashion.

The business process behavior and interaction dynamics might have been specified in a use case model and/or set of related user stories, or even an analysis-level business process model (using BPMN, UML activity diagrams or an equivalent notation). [45]

Problem. How can a client initiate a processing action that causes the server-side application state to advance?

How can API clients and API providers share the responsibilities required to execute and control business processes and their activities? More specifically:

- How can API clients ask an API provider to take over certain functions that represent business activities of varying granularities, from atomic activities to subprocesses to entire processes, but still own the process state ("Frontend BPM")?
- How can API clients initiate, control and follow the asynchronous execution of remote business processes (including subprocesses and activities) exposed and owned by an API provider ("Business Process Management (BPM) services")?

A canonical example process from the insurance domain is claim processing, with activities such as initial validation of a received claim form, fraud check, additional customer correspondence, decision, payment/settlement, and archiving. Instances of this process can live for days to months or even years. Process instance state has to be managed; some parts of the processing can run in parallel whereas others have to be executed one by one sequentially. When dealing with such complex domain semantics, the control and data flow depends on a number of decisions. Multiple systems and services might be involved along the way, each exposing one or more APIs. Other services and application frontends act as API clients. The process instances and the state ownership can lie with the API client (*Frontend BPM*), with the API provider (*BPM services*), or be shared.

Forces. The following specific forces have to be resolved when representing business processes and their activities as API operations, or, more generally speaking, updating provider-side application state:

- Service granularity
- Consistency
- Dependencies on state changes being made beforehand, which may collide with other state changes (e.g., transactions).
- Networking efficiency vs. data parsimony (message sizes)
- Workload management

Time management and reliability also qualify as forces of this pattern; these design concerns are discussed in the pattern *State Creation Operation*.

Details. *Service granularity.* Large services may contain complex and rich state information, updated only in a few transitions, while smaller ones may be simple but chatty in terms of their state transitions.

[72] We call state-preserving processing roles *functions* (as they just get some work done on behalf of a client) and state changing ones *operations* (as they become active because the client hands in some data, which is then processed and stored).

Consistency. Process instances are often subject to audit; depending on the current process instance state, certain activities must not be performed. Some activities have to be completed in a certain time windows (because they require resources that have to be reserved and then allocated). When things go wrong, some activities might have to be undone to bring the process instance and backend resources (e.g., business objects in databases) back into a consistent state.

Dependencies on state changes being made beforehand. State-changing API operations may collide with other state changes (e.g., system transactions triggered by other API clients, by external events in downstream systems, or by provider-internal batch jobs).

Networking efficiency vs. data parsimony (message sizes). One can reduce the weight of the message payload and only send the delta/difference w.r.t. the previous report (an incremental approach). See *State Creation Operation* (Section 4.3) for more information about this force.

Workload management. See *Computation Function* (Section 4.2) for description of this force.

Non-solution. One could decide to ban provider-side application state entirely. This is only realistic in trivial application scenarios such as pocket calculators (not requiring any storage) or simple translation services (working with static data). One could also decide to expose stateless operations and transfer state to and from the endpoint every time. The *Client Session State* pattern in [11] describes the pros and cons of this approach (and the REST principle of hypertext as the engine of application state[73] promotes it). While it scales well, it may introduce security threats and, if state is large, cause performance problems. Client programming becomes more flexible but also more complex and risky; auditability suffers; for instance, how to guarantee that all execution flows are valid (e.g., "order->pay->deliver->return->refund"), preventing "order->deliver(->pay)", "order->deliver->pay->refund", and other fraudulent sequences?.

Solution. Introduce an operation in an API endpoint (typically a *Processing Resource*, or an *Information Holder Resource*) that combines client input and current state to trigger a provider-side state change `f: (in,S) -> (out,S')` (a.k.a. "update action required").

Pair a *Command Message* with a *Document Message* (two "Enterprise Integration Patterns" [19]) to describe the input and the desired action and receive an acknowledgment or result.

How it works. In a business process-like context, for instance claims processing or order management, an API operation may realize a single business activity in a business process or even wrap the complete execution of an entire process instance on the provider side. In this case, calls to this operation trigger one or more instances of the *Business Transaction* pattern described in "Patterns of Enterprise Application Architecture" [11].

When multiple *State Transition Operations* are offered by a *Processing Resource*, the API gives explicit control to the internal processing states so that the client may cancel the execution, track its progress and influence its outcome. This basic principle is shown in Figure 11.

[73] https://en.wikipedia.org/wiki/HATEOAS

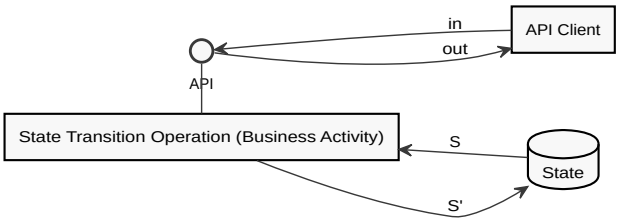


Figure 11: State Transition Operations are stateful, both reading and writing provider-side storage.

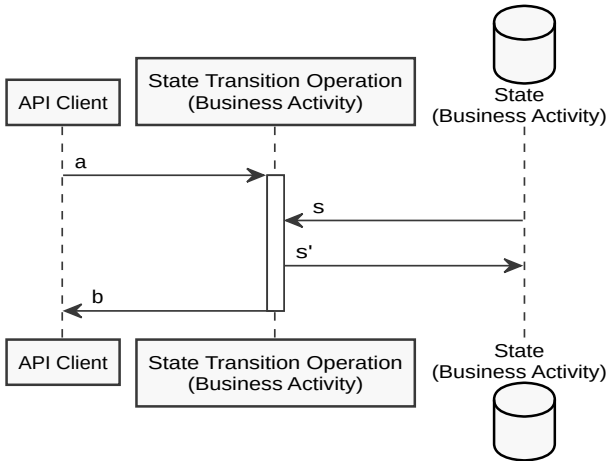


Figure 12: State Transition Operations are stateful, both reading and writing provider-side storage. Process instances have an identity and a lifecycle (see Figure 13).

Single activities can be responsible for any of the following fine-grained process control action primitives:

- `prepare`
- `start`
- `suspend`/`resume`
- `cancel`
- `undo`
- `restart`
- `cleanup`

Given an asynchronous nature of the business activity execution and client-side process ownership (in frontend BPM), it should also be possible to receive the following events as *State Transfer Operations* (see Figure 13):

- `completed` (or, in more detail, `finished`/`failed`/`aborted`)
- `stateChanged`

Prepare (a.k.a. initialize): This primitive allows clients to prepare the execution of the activity by transferring the required information, which could be validated by the provider. Depending on the complexity of such information, initialization may involve a single call or a more complex conversation as illustrated in the *State Creation Operation* pattern. Once all information has been

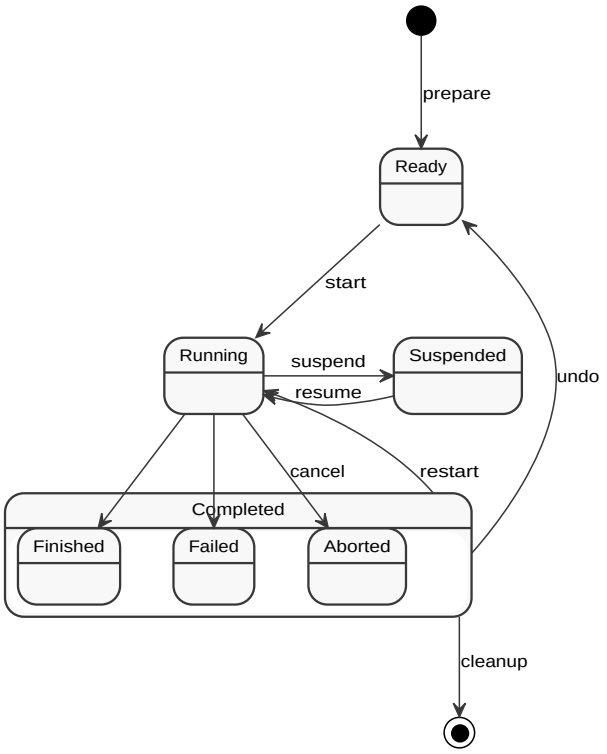


Figure 13: A state machine featuring common transition primitives. These common primitives can be refined in domain-specific process models.

provided the activity is ready to start. This primitive can also be mapped to the sibling pattern *State Creation Operation*.

Start: This primitive allows clients to explicitly start the execution of an activity, which has been initialized and it is ready to start. The state of the activity turns into "running".

Suspend/resume: These primitives allow clients to pause and later resume the execution of a running activity. Suspending a running activity may free execution resources within the provider.

Cancel: This primitive allows clients to interrupt the execution of the activity and abort it in case they are no longer interested about its results.

Undo: This primitive allows compensating the actions performed by the activity, effectively reverting the state of the system back to its original one, before the activity was started. It may not always be possible to do so, especially when activities provoke side-effects that impact the outside of the API provider.

Restart: This primitive allows clients to retry the execution of a failed or aborted activity.

Cleanup: This primitive removes any state associated with completed/failed or aborted activities. The activity identifier is no longer valid.

In Frontend BPM, API clients own the process instance state and have to inform the API provider about the following two types of events (when exposing BPM Services, the other event notification direction is required):

Completed (and/or failure/abortion notification): Once the execution of the activity finishes, affected parties should be notified of its successful or failed completion so that they can retrieve its output.

State changed (a.k.a. getState/notifyStateChange): For monitoring and tracking the progress of the activity, a client might want to fetch the current state of the activity; all affected parties can be notified when a state transition occurs.

State Transition Operations change the business activity state on the API provider side; the complexity of their pre- and postconditions as well as invariants varies, depending on the business and integration scenario at hand. Medium to high complexities of these rules are common in many application domains and scenarios. This behavior must be specified in the *API Description* [28].

Consciously decide where to compose: Frontend BPM often uses a Web frontend as API client, BPM as a Service yields composite services (i.e., *Processing Resource* exposing coarse-grained *State Transition Operations*). Other options are to a) introduce an API Gateway[74] as a single integration and choreography coordination point or b) choreograph services in fully decentralized fashion via peer-to-peer calls and/or event transmission.

From a message structure point of view *State Transition Operation* instances can be fine-grained as well as coarse-grained. Their request message representations vary greatly in their complexity. We present the available structural design space (i.e., types of representation elements) in [52].

Instances of this pattern can be composed to cover subprocesses or entire business processes. If this is done, context information necessary for logging and debugging should also be propagated (for instance, by applying the pattern *Context Representation*).

Many *State Transition Operations* are transactional internally. Operation execution should be governed and protected by a transaction boundary that is identical to the API operation boundary (while this should not be visible to the client on the technical level, it is ok to disclose it in the API documentation due to the consequences for composition, see below). The transaction can either be a *system transaction* following the ACID[75] paradigm [49] or a saga[76], roughly corresponding to compensation-based business transactions[77]. If ACID is not an option, the BASE principles or try-cancel-confirm (TCC) [33] can be considered; a conscious decision between strict and eventual consistency is required, and a locking strategy also has to be decided upon.

Let the processing of the *State Transition Operation* appear to be idempotent, for instance by preferring absolute updates over incremental ones (e.g., "set value of x to y" is easier to process with consistent results vs. "increase value of x by y" which could lead to data corruption if the event gets duplicated/resent). *Idempotent Receiver*[78] in "Enterprise Integration Patterns" [19] provides further advice.

It should be considered to add compliance controls and other security means such as Attribute-Based Access Control (ABAC),

[74] https://microservices.io/patterns/apigateway.html
[75] https://en.wikipedia.org/wiki/ACID
[76] http://microservices.io/patterns/data/saga.html
[77] https://en.wikipedia.org/wiki/Compensating_transaction
[78] https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html

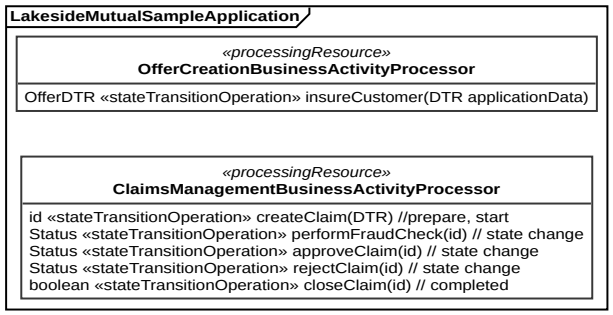


Figure 14: Two examples of State Transition Operations: coarse-grained BPM service and fine-grained Frontend BPM process execution

for instance based on an *API Key* [40] or a stronger authentication token, to the API operation/endpoint may degrade performance.

Variants. There are two quite different types of update, *Full Overwrite (a.k.a.Replacement)* and *Partial Change (a.k.a. Incremental Update)*. Full overwrites/replacements can be processed without accessing current state (and can therefore be seen as instances of the sibling pattern *State Creation Operation*); incremental change typically requires read access to state (as described in this pattern).

Events can either contain absolute new values (*Full Report*) or, as *Delta Reports*, communicate the changes since the previous event (identified by a *Correlation Identifier* or indirectly by timestamp and entity identifier).

With HTTP-based APIs, *Full Overwrite* is typically exposed with the `PUT` method, while *Partial Change* can be achieved with `PATCH`.

Example. The activity "proceed to checkout and pay" in an online shop illustrates the pattern in an order management process. "Add item to shopping basket" then is an activity in the "product catalog browsing" subprocess. These operations do change provider-side state, they do convey business semantics, and they do have nontrivial pre- and postconditions as well as invariants (for instance, "do not deliver the goods and invoice the customer before the customer has checked out and confirmed the order").

The following example from the insurance domain illustrates the two extremes of the pattern; see Figure 14. Offers are created in a single-step operation; claims are managed step-by-step, causing incremental state transitions on the provider side. Some of the primitives from Figure 13 are assigned to *State Transition Operations* in the example.

Implementation hints. Architects and developers that decide to apply this pattern may take the following advice into consideration:

- Apply process-oriented analysis and design techniques such as event storming[79] or use case walkthroughs to carve out events (activity triggers), commands (corresponding to business activities), data entities (business items) and event flows (business processes).
- Consider state machines, supported in middleware such as Spring Integration[80], to keep track of endpoint-internal application state. Also consider workflow languages and engines; one of many offerings is the BPMN engine from Camunda.
- Consider the use or process mining [41] and other techniques from the Business Process Management (BPM) community to monitor and improve the execution of the process flows (for full or partial process execution).
- When implementing the pattern in HTTP resource APIs, use the PATCH (for partial updates) or PUT methods (for full replacements). Use hyperlink formats such as HAL or JSON-LD when referencing previous or subsequent processing steps.

Consequences.

Resolution of forces.

+ Networking efficiency vs. data parsimony (message sizes): A RESTful API design can use state transfers from clients to providers and resource designs to come up with a suited balance between expressiveness and efficiency.
+ Service granularity: *State Transition Operations* can accommodate both smaller and larger "service cuts" [14] and therefore promote agility and flexibility.
+ Consistency: *State Transition Operations* can and must handle business and system transaction management (internally).

– Dependencies on state changes made beforehand may collide with other state changes (see explanation of force above).
– Workload management: stateful *State Transition Operations* cannot scale easily, and endpoints featuring such operations cannot be relocated to other nodes seamlessly (for instance, in when deploying to clouds and striving for IDEAL[81] cloud application properties).

Further discussion. Exposing API operations with business semantics such as business activity/transaction is a key principle and tenet in SOAs and their microservice implementations. A good test is to show the API documentation (e.g., operation name, parameter names/types/descriptions, pre- and postconditions) to business stakeholders a.k.a. domain experts who are not computer scientists or trained in software engineering practices: if such reviewers are able to digest the specification, comment on it and request changes, chances are that the "business alignment" property of the interface is met.

Idempotence is good for fault resiliency and scalability, as mentioned above. But it is often not clear/easy how to achieve it: for instance, one can send "new value is n" message rather than "value of x has increased by one". However, the picture gets more complex in more advanced business scenarios such as order management and payment processing. See coverage of the topic in [10] and [19].

[79] https://www.eventstorming.com/

[80] https://spring.io/projects/spring-integration

[81] The acronym IDEAL (isolated state, distribution, elasticity, automated management, loose coupling) is introduced in [10].

Performance and scalability are primarily driven by technical complexity of the API operation, and the amount of backend processing required in its implementation, the contention to concurrently access shared data, and the resulting IT infrastructure workload (remote connections, computations, disk I/O, CPU energy consumption).

Known Uses. The Application Layer[82] of the Domain-Driven Design Sample Application[83] implements (local) interfaces for two instances of this pattern: `BookingService.java` and `CargoInspectionService.java`.

PayPal has the notion of controller resources[84] and M. Nygard also recommends this behavior- and reponsibility-oriented pattern when talking about activity sets and process services[85]. Note that Nygard considers entity services an anti pattern in all cases, unlike other authors who argue that context matters[86].

Advocates of workflow engines recommend lightweight microservice compositions as well, for instance to decouple steps with an event command transformation[87].

A large body of *State Transfer Operations* can be found in enterprise settings (just to show how common the pattern is):

- Most of the 1000+ services in the core banking SOA featured in [5] qualify as known uses of this pattern, for instance, money transfers.
- The same holds for the business and application services in the telecommunications order management SOA presented in [48]; for instance, the scheduling of a technician visit when customers relocate or upgrade their telephony services is part of a medium complex business process (or workflow).
- The Swiss land register Terravis uses *State Transition Operations* in its backend infrastructure, e.g., for managing mortgages in a position keeping service (e.g., transfer mortgage from one depot to another) or for moving forward in a business process instance (e.g., submit digitally signed document) [29].

Related Patterns. The patterns differs from its siblings like this: A *Computation Function* does not touch the provider side application state (read or write) at all; a *State Creation Operations* only writes to it (in append mode). Instances of *Retrieval Operation* read, but do not write it; *State Transition Operation* instances both read and write the provider-side state. *Computation Function* and *Retrieval Operation* pull information from the provider; *State Creation Operations* (such as instances of its *Event Notification Operation* variant) push updates to the provider. *State Transition Operations* may push and/or pull. A *State Transition Operation* may refer to a provider-side state element in its request message (for instance, an order id or serial number of a staff member); *State Creation Operations* usually do not do this. A single API endpoint may, but not necessarily should apply more than one of these patterns; command-query separation[88] is a principle from object-oriented programming also eligible on the architectural level.

State Transition Operations can be seen to trigger and/or realize the *Business Transactions* [11]. Instances of this pattern may participate in long running and therefore stateful *conversations* [17]. They can use and go along with one or more of the RESTful conversation patterns from [35]. For instance, one may want to consider factoring out the state management and the computation part of the pattern into separate services. *Conversation patterns* or choreographies and/or orchestrations may then define the valid combinations and execution sequences of these services.

State Transition Operations are often exposed in *Community APIs*; if this is done, they can be protected with an *API Key* [40] and their usage can be governed with a *Service Level Agreement* [40].

In Domain-Driven Design (DDD) [9], the *Aggregate* and *Entity* patterns have related semantics (i.e., they represent groups of domain concepts that have an identify and a lifecyle). Hence, these patterns can help to identify *State Transition Operation* candidates during endpoint identification. It is important not to expose the entire domain as *Published Language* on the API level because this creates an undesired tight coupling between the API clients and the provider-side API implementation.

Other Sources. The is a large body of literature on BPM(N) and workflow management that introduces concepts and technologies to implement stateful service components in general and *State Transition Operations* in particular, for instance [26] [27] [3] [12].

In Responsibility-Driven Design (RDD), *State Transition Operations* correspond to *coordinators* and *controllers* that are encapsulated as *service providers* made accessible from remote with the help of an *interfacer* as described in [44].

The seven-step service design method[89] in the Software/Service/API Design Practice Repository (DPR) suggests to call out endpoint roles and operation responsibilities such as *State Transition Operation* when preparing candidate endpoint lists and refining them.

5 CONCLUSIONS AND OUTLOOK

The knowledge captured in this paper already has been used as guidance for making architectural decisions in industry projects. Our patterns are applicable not only to microservice APIs, but also to any remote API based on plain document messages rather than stateful protocols or remote objects. Both synchronous APIs using direct HTTP exchanges and asynchronous, queue-based ones are in scope.

Selected patterns are implemented in the *Lakeside Mutual*[90] scenario and sample application. Lakeside Mutual is a fictitious insurance company that implemented its core business capabilities for customer, contract, and risk management as a set of microservices with corresponding application frontends. The emerging *Microservice Domain Specific Language (MDSL)*[91] features all responsibility patterns as endpoint or operation decorators; it also integrates similar role stereotype information for the entire API and on the level of message representation elements. Our new Software/Service/API

[82] https://github.com/citerus/dddsample-core/tree/master/src/main/java/se/citerus/dddsample/application
[83] http://dddsample.sourceforge.net/characterization.html
[84] https://github.com/paypal/api-standards/blob/master/patterns.md#controller-resources
[85] http://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/
[86] https://icwe2020.webengineering.org/keynotes/#academy
[87] https://www.infoq.com/articles/microservice-event-choreographies
[88] https://martinfowler.com/bliki/CommandQuerySeparation.html
[89] https://github.com/socadk/design-practice-repository
[90] https://github.com/Microservice-API-Patterns/LakesideMutual
[91] https://microservice-api-patterns.github.io/MDSL-Specification/tutorial

Design Practice Repository (DPR)[92] features the responsibility patterns in Step 5 of its stepwise service design method. The blog post "MAP Retrospective and Outlook"[93] reflects on the evolution of our pattern language since 2017.

As part of our future work, we consider to cover service implementation details in addition to API contract design. Candidate patterns include *Guard Resource* (a wrapper around a backend system, *Ground Resource* (a service not having any outbound dependencies) and *Composed Resource* (representing orchestrations and choreographies). We also consider to extend our pattern collection with additional structural, behavioral, and process patterns.

ACKNOWLEDGMENTS

We want to thank the EuroPLoP shepherds and writers' workshop participants that provided constructive feedback since 2017, students and members of our professional networks who helped to investigate public Web APIs, donated pattern candidates and known uses, and reviewed early drafts of pattern candidates and language structure. The work of Olaf Zimmermann and Mirko Stocker on MDSL and DPR is supported by the Hasler Foundation. The work of Cesare Pautasso and Uwe Zdun was supported by the API-ACE project, funded by SNF project 184692 and FWF (Austrian Science Fund) project I 4268.

REFERENCES

[1] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly.
[2] Deepak Alur, Dan Malks, and John Crupi. 2013. *Core J2EE Patterns: Best Practices and Design Strategies* (2nd ed.). Prentice Hall.
[3] Jesus Bellido, Rosa Alarcón, and Cesare Pautasso. 2013. Control-Flow Patterns for Decentralized RESTful Service Composition. *ACM Transactions on the Web (TWEB)* 8 (December 2013), 5:1–5:30. https://doi.org/10.1145/2535911
[4] Walter Berli, Daniel Lübke, and Werner Möckli. 2014. Terravis – Large Scale Business Process Integration between Public and Private Partners. In *Lecture Notes in Informatics (LNI), Proceedings INFORMATIK 2014*, Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull (Eds.), Vol. P-232. Gesellschaft für Informatik e.V., Gesellschaft für Informatik e.V., 1075–1090.
[5] Michael Brandner, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. Web services-oriented architecture in production in the finance industry. *Informatik-Spektrum* 27, 2 (2004), 136–145. https://doi.org/10.1007/s00287-004-0380-2
[6] Kyle Brown and Bobby Woolf. 2016. Implementation Patterns for Microservices Architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs (PLoP '16)*. The Hillside Group, USA, Article Article 7, 35 pages.
[7] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing.* Wiley.
[8] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley Professional. http://www.servicedesignpatterns.com/
[9] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley.
[10] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer.
[11] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley.
[12] Alessio Gambi and Cesare Pautasso. 2013. RESTful Business Process Management in the Cloud. In *5th ICSE International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2013)*. San Francisco, CA, USA.
[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.
[14] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 185–200.
[15] Robert Hanmer. 2007. *Patterns for Fault Tolerant Software.* Wiley.
[16] Carsten Hentrich and Uwe Zdun. 2011. *Process-Driven SOA: Patterns for Aligning Business and IT.* Auerbach Publications.
[17] Gregor Hohpe. 2007. Conversation Patterns: Interactions between Loosely Coupled Services. In *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany.
[18] Gregor Hohpe. 2007. SOA Patterns: New Insights or Recycled Knowledge? Online article. https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf
[19] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley.
[20] Nicolai Josuttis. 2007. *SOA in Practice: The Art of Distributed System Design.* O'Reilly.
[21] Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. 2011. Compliance by design–Bridging the chasm between auditors and IT architects. *Computers & Security* 30, 6 (2011), 410–426.
[22] Stefan Kapferer and Olaf Zimmermann. 2020. Domain-driven Service Design - Context Modeling, Model Refactoring and Contract Generation. In *Proc. of the 14th Advanced Summer School on Service-Oriented Computing (SummerSOC'20) (to appear)*. Springer CCIS.
[23] Ralph Kimball and Margy Ross. 2002. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* (2nd ed.). John Wiley.
[24] Dirk Krafzig, Karl Banke, and Dirk Slama. 2004. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series).* Prentice Hall.
[25] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. https://martinfowler.com/articles/microservices.html
[26] Frank Leymann and Dieter Roller. 2000. *Production workflow - concepts and techniques.* Prentice Hall.
[27] Frank Leymann, Dieter Roller, and Marc-Thomas Schmidt. 2002. Web services and business process management. *IBM Syst. J.* 41, 2 (2002), 198–211. https://doi.org/10.1147/sj.412.0198
[28] Daniel Lübke, Olaf Zimmermann, Mirko Stocker, Cesare Pautasso, and Uwe Zdun. 2019. Interface Evolution Patterns - Balancing Compatibility and Extensibility across Service Life Cycles. In *Proc. of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*.
[29] Daniel Lübke and Tammo van Lessen. 2016. Modeling Test Cases in BPMN for Behavior-Driven Development. *IEEE Software* 33, 5 (Sept.-Oct. 2016), 15–21.
[30] Bertrand Meyer. 1997. *Object-oriented Software Construction (2nd Ed.).* Prentice-Hall.
[31] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. 2020. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Softw.-Intensive Cyber Phys. Syst.* 35, 1 (2020), 3–15. https://doi.org/10.1007/s00450-019-00407-8
[32] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly.
[33] Guy Pardon and Cesare Pautasso. 2011. *Towards Distributed Atomic Transactions over RESTful Services.* Springer, 507–524. https://doi.org/10.1007/978-1-4419-8303-9_23
[34] Guy Pardon, Cesare Pautasso, and Olaf Zimmermann. 2018. Consistent Disaster Recovery for Microservices: the BAC Theorem. *IEEE Cloud Computing* 5, 1 (12 2018), 49–59. https://doi.org/10.1109/MCC.2018.011791714
[35] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany.
[36] Cesare Pautasso and Olaf Zimmermann. 2018. The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software* 35 (January/February 2018), 93–98. https://doi.org/10.1109/MS.2017.4541049
[37] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software* 34, 1 (2017), 91–98. https://doi.org/10.1109/MS.2017.24
[38] Chris Richardson. 2016. Microservice Architecture. http://microservices.io. (2016).
[39] Chris Richardson. 2018. *Microservices Patterns.* Manning.
[40] Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2018. Interface Quality Patterns - Communicating and Improving the Quality of Microservices APIs. In *Proc. of the 23nd European Conference on Pattern Languages of Programs (EuroPLoP '18)*.
[41] Wil M. P. van der Aalst. 2011. *Process Mining: Discovery, Conformance and Enhancement of Business Processes.* Springer.
[42] Vaughn Vernon. 2013. *Implementing Domain-Driven Design.* Addison-Wesley Professional.
[43] 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.
[44] Rebecca Wirfs-Brock and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations.* Pearson Education.
[45] Olaf Zimmermann. 2009. *An architectural decision modeling framework for service-oriented architecture design.* Ph.D. Dissertation. University of Stuttgart, Germany.

[92] https://github.com/socadk/design-practice-repository
[93] https://ozimmer.ch/patterns/2020/04/29/MAPRetrospective.html

http://elib.uni-stuttgart.de/opus/volltexte/2010/5228/

[46] Olaf Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (Mar.-Apr. 2015), 26–29. https://doi.org/10.1109/MS.2015.37

[47] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[48] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. 2005. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. (2005), 301–312.

[49] Olaf Zimmermann, Jonas Grundler, Stefan Tai, and Frank Leymann. 2007. Architectural Decisions and Patterns for Transactional Workflows in SOA. In *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings (Lecture Notes in Computer Science)*, Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan (Eds.), Vol. 4749. Springer, 81–93. https://doi.org/10.1007/978-3-540-74974-5_7

[50] Olaf Zimmermann, Daniel Pautasso, Cesare Lübke, Uwe Zdun, , and Mirko Stocker. 2019. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proc. of the European Conference on Pattern Languages of Programs (EuroPLoP '19)*.

[51] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to Microservice API Patterns (MAP). In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019) (OpenAccess Series in Informatics (OASIcs))*, Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh (Eds.), Vol. 78. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[52] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proc. of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*. ACM, Article 27, 36 pages. https://doi.org/10.1145/3147704.3147734