Towards Concern-Oriented Microservice Architecture
Hugo Monfleur, Philippe Merle

To cite this version:
Hugo Monfleur, Philippe Merle. Towards Concern-Oriented Microservice Architecture. 2023 International Conference on Microservices, Universita di Pisa; Microservices Community, Oct 2023, Pise, Italy. hal-04201189

HAL Id: hal-04201189
https://inria.hal.science/hal-04201189
Submitted on 9 Sep 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Towards Concern-Oriented Microservice Architecture

Hugo Monfleur · Philippe Merle
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
✉ hugo.monfleur@inria.fr

Abstract. Existing microservice architecture languages do not allow for the distinction and expression of intertwined cross-cutting architectural concerns, i.e. aspects that affect several microservices, without the possibility of being encapsulated in any of them. To address this problem, this paper proposes the Concern-Oriented Microservice Architecture metamodel where cross-cutting architectural concerns are first-class citizens and show how the worldwide-used Docker Compose language can be equipped with new concern-oriented constructions. This extension is applied to the TeaStore microservice reference application.

1 Introduction

Any microservice application has to deal with cross-cutting architectural concerns, i.e. aspects that affect several microservices, without the possibility of being encapsulated in any of them. A particular concern can then be defined formally as a predicate on units that indicates whether or not a given unit pertains to that concern. The problem lies in the fact that these concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and can result in either scattering (code duplication), tangling (significant dependencies), or both. Separation of concerns, probably coined by Edsger W. Dijkstra in and consisting in the study of one’s subject matter in isolation for the sake of its own consistency, is then vital for concision, comprehensiveness, maintainability, and evolvability of microservice architectures.

To help get a better grip of what a concern is and how diverse they can be in the context of microservice applications, we introduce three examples. Firstly, each application is composed of microservices. Some are dedicated to implementing business logic while others implement non-functional properties such as logging, monitoring, distributed tracing, persistence, security, or fault-tolerance. All of these relate to high level architecture decisions that determine the value of properties of microservices in regard to a particular aspect. As such they are cross-cutting architectural concerns. Secondly, various design patterns have been identified in the literature (e.g. Service Registry, API Gateway, Circuit Breaker, or Distributed Tracing. Here again, these patterns are cross-cutting architectural concerns. Finally, each application has its own patterns for naming microservice identities, images, environment variables, etc. To the best of our knowledge, there is no microservice architecture language offering constructions for dealing with such cross-cutting architectural concerns.


© 2023 The Authors. This is document is shared under the terms of the Creative Commons Attribution License (CC-BY 4.0), which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
This paper argues for making all intertwined cross-cutting architectural concerns first-class citizens of microservice architecture languages. This implies equipping microservice architecture languages with new concern-oriented constructions. In order to be language-agnostic, we propose a Concern-Oriented Microservice Architecture metamodel where cross-cutting architectural concerns are first-class citizens. In order to be concrete, we choose to extend the Docker Compose language as it is largely used in both academia and industry. A recent search on GitHub\(^2\) returns more than 672k Docker compose files. Then we apply this concern-oriented extension of Docker Compose to TeaStore, a micro-service reference application\(^3\)[8]. We chose this use case for validating our proposal as it has been quoted in over 100 scientific publications\(^4\) and is also used in industry, e.g. by the Cisco Full Stack Observability Lab\(^5\).

The rest of the paper is organized as follows. Section 2 presents our motivating example, the TeaStore application, and emphasizes its multiple scattered cross-cutting architectural concerns. Section 3 overviews the principles of our Concern-Oriented Microservice Architecture contribution, then revisits TeaStore for making its concerns first-class citizens of Docker Compose, and sketches a toolchain for dealing with concern-oriented microservice architectures. Section 4 reviews some related work. Finally, Section 5 concludes and discusses future perspectives.

### 2 Motivating Example

TeaStore is a **micro-service reference application for benchmarking, modeling, and resource management research**\(^8\). This application, developed as a real-life application with best practices in mind, is an **online shop for tea and tea-related utilities**\(^8\). TeaStore is composed of seven microservices: six are represented in Figure\(^1\) and the seventh is a database dedicated to the Persistence microservice.

![Figure 1: Architecture of the TeaStore application](https://raw.githubusercontent.com/DescartesResearch/TeaStore/master/examples/docker/docker-compose_default.yaml)

Listing\(^1\)is the Docker Compose file of the default architecture of the TeaStore application\(^6\).

```
version: '3'
services:
  registry:
    image: descartesresearch/teastore-registry
    expose:
      https://github.com/search?q=path%3A**%2Fdocker-compose*.y*ml&type=code
      https://github.com/DescartesResearch/TeaStore
      https://scholar.google.fr/scholar?cites=7828615202992375775
      https://fso.cisco-one.com/lab-environment/
```
```yaml
- 8080

image: descartesresearch/teastore-db
expose: - "3306"
ports: - "3306:3306"

persistence:
image: descartesresearch/teastore-persistence
expose: - "8080"

environment:
HOST_NAME: "persistence"
REGISTRY_HOST: "registry"

recommender:
image: descartesresearch/teastore-recommender
expose: - "8080"

environment:
HOST_NAME: "recommender"
REGISTRY_HOST: "registry"

webui:
image: descartesresearch/teastore-webui
expose: - "8080:8080"

environment:
HOST_NAME: "webui"
REGISTRY_HOST: "registry"

ports: - "8080:8080"
```

Listing 1: Default Docker Compose of the TeaStore Application

Even if this motivating example seems simple, it contains seven intertwined cross-cutting architectural concerns. The first concern is related to the identity (e.g. registry) of the six business microservices shown in Figure 1 and is scattered across lines 3, 13, 22, 29, 36, and 43. The second concern is related to the Persistence composite microservice, which is split into both strongly coupled persistence and db microservices (Lines 7-21). The third concern is the Service Registry design pattern, scattered across lines 3, 19, 28, 35, 42, and 49. persistence, auth, image, recommender and webui microservices can discover each others via the registry microservice. The fourth concern is related to the image naming pattern, scattered across lines 4, 8, 14, 23, 30, 37, and 44. We can notice that all seven image names share the same prefix descartesresearch/teastore- and their suffix is always the identity of their owning microservice. The fifth concern is the host name regularity scattered across lines 18, 27, 34, 41, and 48. The HOST_NAME environment variable is always assigned to the identity of its owning microservice. The sixth concern is the exposed container network port regularity scattered across lines 5-6, 15-16, 24-25, 31-32, 38-39, and 45-46. The six business services expose the same container network port, aka 8080. Finally, the seventh concern is related to public network ports exposed by the TeaStore application, and is scattered across lines 11-12 and 50-51. Only ports 3306 of db and 8080 of webui are publicly accessible.

3 Concern-Oriented Microservice Architecture

3.1 Principles
As illustrated in Figure 2a in a classical Microservice Architecture (MSA) approach, each Application owns a none-empty set of microservices. Each Microservice has a unique distinct identity msid (e.g.
registry) and owns a non-empty set of properties. Each Property has a unique distinct name (e.g. image) and a value (e.g. descartesresearch/teastore-registry).

![Diagram of MSA and COMSA Metamodels](image)

**Figure 2: Microservice Architecture vs Concern-Oriented Microservice Architecture**

As illustrated in Figure 2b, the metamodel of our Concern-Oriented Microservice Architecture (COMSA) approach is subtly different. Each Application owns a non-empty set of concerns, making concerns first-class citizens. Each Concern has a unique distinct identity cid and references a non-empty set of microservices. A microservice can be shared by several concerns. Each Microservice still has the msid attribute but references a set of properties instead of owning its properties. Thus a property can be shared by several microservices. Finally, Property is unchanged.

The abstract concepts of our COMSA approach must be concretely implemented in target existing microservice architecture languages. The next section illustrates how the COMSA concepts can be reflected by new concern-oriented constructions in the Docker Compose language.

### 3.2 Motivating Example Revisited

Listing 2 revisits the motivating TeaStore example by making its seven concerns first-class citizens thanks to the new concerns construct in Line 2.

```plaintext
1 version: '3'
2 concerns:
3   BusinessServices:
4     services:
5       (registry, persistence, auth, image, recommender, webui):
6 DescartesResearchImages:
   services:
```

4
The first concern is encoded by BusinessServices at Lines 3-5. Line 5 shows another new construct to define sets of microservices. DescartesResearchImages at Lines 6-9 encodes the fourth concern previously identified. Lines 8-9 encode the fact that all microservices of the TeaStore application (ALL keyname) share a common image naming pattern. The {SID} expression refers to the microservice identity. RegistryDesignPattern at Lines 10-18 mainly implements the third concern at Line 18 but also the fifth and sixth concerns, Line 17 and Lines 12-14 respectively. In fact, both regularities are strongly coupled with the service registry design pattern. Line 12 shows how to refer to all the microservices of another concern (here BusinessServices) and Line 15 illustrates the difference operation between sets: BusinessServices \ registry equals to (persistence, auth, image, recommender, webui). The second concern is implemented by Persistence at Lines 19-27. The expression \{db.expose[0]\} accesses the first item of the expose array property of the db microservice. Finally, the seventh concern is encoded by Ports at Lines 28-32. Both \{expose[0]\} expressions access the first item of the expose property of the current microservice (db then webui).

To summarize, our concern-oriented extension of the Docker Compose language provides:

- the new concerns keyword for making concerns first-class citizens of Docker Compose,
- a new notation to manipulate sets of microservices, and
- a powerful notion of expressions to avoid redundancy of property values.

This extension provides multiple benefits such as clear separation of microservice concerns for better comprehension, concision (32 lines instead of 51 lines, i.e. a gain of 37% for the TeaStore application), ease of maintainability, and evolvability.
3.3 COMSA Toolchain

Figure 3 shows the current architecture of our concern-oriented microservice architecture toolchain.

Figure 3: Concern-Oriented Microservice Architecture Toolchain

**Concerns Verification** targets the verification and validation of COMSA files, i.e. syntactical and semantics checking of concern declarations, microservice sets, and property value expressions. **Concerns Visualization** generates visual concern-oriented diagrams. Hypergraphs are particularly well suited to visualize concern-based applications as illustrated in the right part of Figure 3, i.e. a hypergraph representing the concerns of the TeaStore application. **Compose Generation** generates standard Docker Compose files from COMSA files, i.e. compose the concerns expressed in Listing 2 to generate Listing 1 automatically. **Concerns Extraction** does the reverse job of extracting concerns from any existing Docker Compose files. Here the challenge is to find the right heuristics to automatically build pertinent concerns.

4 Related Work

To the best of our knowledge, no MSA language provides concern-oriented constructions even though some MSA languages deal with specific concerns. LEMMA [4] is a modeling MSA language ecosystem including four viewpoints (Domain, Operation, Service, Technology), each addressing a specific set of stakeholders’ concerns. Silvera [6] is a Domain-Specific Language for modeling MSA including constructions to deal with only five MSA design patterns, i.e. Service Registry, API Gateway, RPC and Messaging Communication Styles, and Circuit Breaker. MicroART [3] provides a MSA recovery metamodel including an attribute to qualify the nature of each microservice, i.e. functional (what we name business), monitoring, system-level management, service orchestration, service brokering, security, service proxy, and data storage. All these can be easily represented by our MSA concerns.

5 Conclusion and Perspectives

The key idea of this paper is to make any cross-cutting architectural concern a first-class citizen of MSA languages. For this, the paper introduces the COMSA approach, its metamodel and its toolchain, proposes a concern-oriented extension of the Docker Compose language, and applies it to the well-known TeaStore microservice reference application [8].
COMSA work is still in its infancy. Perspectives encompass enhancing COMSA principles, metamodel and toolchain, applying COMSA to various large-scale MSA applications such as those of the DeathStarBench benchmark suite, developing a library of reusable cross-cutting architectural concerns, and applying COMSA to other existing MSA languages.

Acknowledgments

This work was partially funded by the French ANR SCALER project.

References


