Business Object Exchange

One microservice is responsible for (multiple) managed data objects (MO) that have to be transmitted to other microservices. This concept describes the exchange of MOs that bases on RESTful webservices as mainly implemented by microservice architectures (Dragoni et al. 2017).

REST uses the HTTP verbs for the basis operations of the CRUD principle (`Create`, `Read`, `Update` and `Delete`). Thus, the creation of instances uses `POST`, while `GET` is used to call for instances and an update uses `PUT` (as well as the parallel creation of multiple instances). Delete uses the descriptive verb `DELETE` (Rodriguez, 2008).

RESTful Webservices are stateless: Requests contain all required information to answer the request and responses can describe links to other resources. Furthermore, responses can be cached. Additionally, webservices are using URIs following a directory structure: A hierarchy of (sub) pathes is extending a root node and query strings should be avoided. Moreover, the data should by human readable, using formats like XML, JSON or both in parallel (Rodriguez, 2008).

Considering the expected loosely coupling of the Heureka! architecture, the communication between microservices should follow the concept of choreography, instead of orchestration (Nikaj and Weske, 2016; Nikaj et al., 2016). Thus, the communication between two services requires only a direct communication channel between both. Therefore, a network of microservices results and there are only dependencies between the directly communicating nodes.

Selection of data

It is a challenge for software developers of microservices to decide which data should be provided in RESTful webservices. Thus, MOs are categorized: (1) Own MOs that are saved and managed by the microservice itself and (2) pulled MOs that have been received from other microservices. Only the first category is relevant, that can be splitted in (1.1) pushed MOs and (1.2) internal MOs. Only the type (1.1) of the pushed MOs are relevant for other microservices.

In some special cases, additional information for pulled MOs are created and managed. If these information should become pushed by a RESTful webservice can be decided by the escalation guideline: A MO has to be always complete in the network. Thus, data extensions are handled only as internal MOs.

Escalation guideline

The guideline defines rules for the change of pushed and pulled MOs. The following table describes a rule and a reasoning for each case:

Case	Rule	Reasoning

Case	Rule	Reasoning
Extend a pushed MO	Pushed MOs can be extended by increasing the version number. Furthermore, the old versions have to be still accessible by the old version numbers.	Conflicts with pulled MOs have to be avoided. Thus, the software developers should be responsible by themselves regarding the robustness of their system. This becomes possible by the versioning of data formats.
Attributes of a pushed MO are altered	Changes can become implemened and have to be provided by a new version number. The previous data description has to be accessible by the previous version number.	Conflicts with pulled MOs have to be avoided. Thus, the software developers should be responsible by themselves regarding the robustness of their system. This becomes possible by the versioning of data formats.
Attributes of a pushed MO will be removed	Attributes can be deleted and have to be provided by a new version number. The previous data description has to be accessible by the previous version number.	Conflicts with pulled MOs have to be avoided. Thus, the software developers should be responsible by themselves regarding the robustness of their system. This becomes possible by the versioning of data formats.
A pulled MO will be extended	A collaboration process with the developers of the providing microservices has to be initiated. If the extension is also required by other services, a new requirement for the providing microservices has to be created. If this is not the case, the new requirement has to be implemented as an internal MO in the receiving microservice.	The technical communication should be as simple and managable as possible. Thus, MOs should not be dispersed on different services.
Attributes of a pulled MO must be changed	A collaboration process with the developers of all receiving microservices has to be initiated. If the change is accepted by the developers of multiple receiving microservices, a new requirement for the providing microservice has to be formulated and a new version should be released. If the change is only required by the originally initiating microservice, a solution has to be implemented in this microservice.	Conflicts with pulled MOs have to be avoided. Thus, the software developers should be responsible by themselves regarding the robustness of their system. This becomes possible by the versioning of data formats.
Attributes of a pulled MO are not required anymore	See previous solution (Attributes of a pulled MO have to be changed)	See reasoning before (Attributes of a pulled MO have to be changed)

Life cycles - Object event system

If the provided data of a microservice is changed, other services may have to react. For example, users are often associated to other data objects. If a user deletes or deactivates his / her account, it is required for other microservices to detect this change and to delete the corresponding association.

Thus the *Object Event System (OES)* ist introduced as ab additional communication layer. A modern message broker ist used to push updates of data to other services that have subscribed for these data. Messages describe the data of

change as well as the operation (`delete` or `update`). Afterwards, receiving microservices can request the data object again, to update all references or delete these references.

The messages have the following format:

```
{
  "sender": "microservice-uuid",
  "action": "action-id",
  "object": "object-uuid",
  "type": "object-type",
  "timestamp": 123456789
}
```

While the attribut `sender` identifies the providing microservice, `action` describes the altering operation. `action` can have four different values: `delete` , `update` , `deactivated` or `activated` . In difference to `deactivated` , `delete` describes the complete deletion of the object. Thus, it will not be possible to request the data object again. `deactivated` means that the object is still saved in the database, but not actively used anymore. The `action` `activated` can be used to reactivate deactivated objects. The attribute `object` identifies the addressed object using an `UUID` . `type` supports the contextualization of the object and describes the type. The unix `timestamp` marks the time the operation has been executed.

It is not required to register the receiving microservice for the RESTful webservices at the providing microservices. Thus, it is not possible for the providing microservice to identify the receiving microservices and to send them messages. A publish-subscriber mechanism addresses this technical challenge. Every receiving microservice registrates itself for specific messages from the providing microservice. Thus, a providing microservice is able to send messages to the receiving microservices.

The open source NATS message broker is used to implement the publish-subscriber mechanism.

Communication

RESTful webservices base on HTTP endpoints accessible by a URI. Thus, to exchange data, microservices need to know the endpoints that have to be well-documented using an OpenAPI Specification. Thus, on this level of communication the microservices know each other.

Security

In the first step, the communication is implemented only between docker containers. Thus, it is possible to separate the network of microservices from the rest of the word.

> The docker containers need to communicate using their internal docker IP addresses. Otherwise the messages would be send through the internet.

Consistency of interfaces

The following guidelines are created to increase the consistency of the interfaces between the microservices.

Considering the general guidelines regarding RESTful interfaces (Rodriguez, 2008), the request of data should be done using `HTTP GET`. Thus, it is still an open question how to specify messages without using a query body, but also without having a complex query string. Therefore, microservices have to provide two parameter for a `GET` request:

First, a filter containing a stringified JSON with descriptions of partial defined entities. These are data containers implementing the same data structures as the managed entities of the microservice. In contrast to the MO, all values of partial defined entities are optional. Furthermore, also the values of the attributes are partially defined (that means only partially matching to existing values). Thus, values for filters can be described.

Additionally, it is required to relate the partial values to each other by the AND and OR relations. Moreover, some unary operations, like equality or a logical NOT are required.

These relations and operations are described by the value encoded n the query as a string. The value string consists of syntactical elements (structures like brackets and operations) as well as the names of the attributes of the partial defined entities. The allowed relations and operations are:

Binary operations (relations between attributes of partial defined entities):

- `_OR_` - logical OR
- `_AND_` - logical AND

Unary operations (addressing the values of the partial defined entities):

- `!` - logical NOT
- `=` - equal
- `<` - smaller
- `>` - larger
- `<=` - less or equal
- `>=` - larger or equal
- `!=` - unequal
- `IN` - The value of the attribute of the addressed entity is in the list of possible values given by the partial defined entity.
- `BETWEEN` - The value of the attribute of the addressed entity is between the two values given by the partial defined entity.
- `LIKE` - The value of the attribute of the addressed entity is similar to the value given by the partial defined entity.

Beispiel: `first_name.LIKE_AND_email.=_AND_(crew.name.LIKE_OR_placeOfResidence.LIKE)`

> User `.` to describe pathes of attributes in the query and the unary operations are always concatinated to a attribute path by `.`.

> Attribute names consisting of multiple words using the cammel case notation, as suggested by the JSON Style Guide of Google (Google, 2007).

> See the implemention of all existing microservice in the REST APIs documentation.

References

(Dragoni et al., 2017)	N. Dragoni et al., “Microservices: yesterday, today, and tomorrow.” Cornell University, 2017.
(Rodriguez, 2008)	A. Rodriguez, “RESTful Web services: The basics,” 2008. [Online]. Available: https://www.ibm.com/developerworks/library/ws-restful/. [Accessed: 17-Mar-2017]
(Nikaj and Weske, 2016)	A. Nikaj and M. Weske, “Formal Specification of RESTful Choreography Properties,” in Web Engineering. ICWE 2016. Lecture Notes in Computer Science, 2016, pp. 365–372.
(Nikaj et al., 2016)	A. Nikaj, S. Mandal, C. Pautasso, and M. Weske, “From Choreography Diagrams to RESTful Interactions,” in Service-Oriented Computing – ICSOC 2015 Workshops. ICSOC 2015. Lecture Notes in Computer Science, vol 9586, 2016, pp. 3–14.
(Google, 2007)	“Google JSON Style Guid - Property Name Format” 2007. [Online]. Available: https://google.github.io/styleguide/jsoncstyleguide.xml#Property_Name_Format/. [Accessed: 21-Oct-2017]