Authorization-aware HATEOAS

Marc Hüffmeyer[1], Florian Haupt[2], Frank Leymann[2] and Ulf Schreier[1]
[1]*Hochschule Furtwangen, Furtwangen im Schwarzwald, Germany*
[2]*Universität Stuttgart, Stuttgart, Germany*

Keywords: REST, Web Services, Authorization, Attribute Based Access Control.

Abstract: The architectural style named Representational State Transfer (REST) is nowadays widely established and still enjoys a growing popularity. One of the core principles of REST is referred as "Hypermedia as the Engine of Application State" (HATEOAS). HATEOAS is one of the foundations of the scalability that RESTful systems provide and enables the decoupling of client and server. But the realization of HATEOAS is challenging, because there is no systematic approach how to enforce the constraint. Therefore, the implementation is mostly up to the developer of a RESTful service. This work describes a new method of how to apply the HATEOAS constraint. We describe a method that systematically enables HATEOAS based on REST API models and the integration of access control mechanisms. In order to avoid unauthorized access attempts and unnecessary network traffic, the resource representations are customized to the requesting subject. References that lead to not accessible resources, are not included in the customized resource representations. Therefore, an attribute based access control mechanism is extended to distinguish between static and dynamic attributes. A 2-phase authorization procedure is introduced that relies on this discrimination and determines the references which must be included in the resource representation. The result is a flexible realization of HATEOAS based on formal models.

1 INTRODUCTION

REST (Fielding, 2000) is an architectural style that describes foundations on how to build highly scalable, distributed systems. The building blocks of REST have been developed as an abstraction of the World Wide Web which can be seen as a primary example of a highly scalable, distributed system. Applications that follow the constraints of the architectural style will benefit from scalabilty, mashup-ability, usability and accessability (Wilde and Pautasso, 2011).

HATEOAS is one of the major constraints of REST and requires that a server is able to send possible application state transitions to the client (Amundsen, 2017). Therefore, it is reasonable to model the state transitions between resources as metadata. Having a metamodel which describes the state transitions enables to exploit the model in order to apply access control. State transitions that must not be performed by the client can be skipped and not included in the response. This helps to increase security, reduce unnecessary network traffic (especially in machine-to-machine communication) and eliminate annoying navigation paths for users. The presented approach uses REST API models and the HATEOAS constraint to integrate fine-grained access control with RESTful services. Instead of providing plain representations of resources in response to a service request, the results of authorization-aware HATEOAS processing are customized representations of resources that respect access rights in advance.

The remainder of this works is organized as follows: the paper first introduces an example scenario which is used throughout the paper in section 2. In section 3 we describe the foundations of authorization-aware HATEOAS, namely REST, hypermedia navigation models, attribute based access control and an access control system for RESTful services named RestACL. In section 4 the main ideas are described on how to build a RESTful system that is authorization-aware. An evaluation of the proposed system is presented in section 5. Finally, related work, conclusion and future work complete this work.

2 EXAMPLE SCENARIO

Industry 4.0 (Lasi et al., 2014) is an area where different subjects like customers, suppliers, providers, op-

Hüffmeyer, M., Haupt, F., Leymann, F. and Schreier, U.
Authorization-aware HATEOAS.
DOI: 10.5220/0006683700780089
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 78-89
ISBN: 978-989-758-295-0

erators and smart machines act together on the same resources. Imagine a factory that builds a dedicated product and offers the customer to parameterize the product even if it is already in production. A automotive factory might be an example where customers can decide to change the interior even if body parts and the engine are already in production. From a REST oriented point of view, the product is a resource and the parameterizable parts are subresources of that resource. Figure 1 depicts a state chart of a single resource and shows the state transitions for this resource. Inside the boxes the particular state name is mentioned. The transitions between the boxes indicate the operations that can be performed to create or update the resource. Note that state charts and resources in a real world application are much more complex and we use the same simplified chart for all resources (cf. Table 1) in order to focus on the main

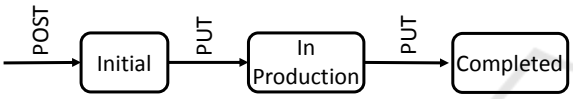

Figure 1: Simplified state chart for production state.

For example, a customer can order a product by sending a creation request to the factory (e.g. by sending a POST request to the address *http://example.org/products*). The product resource is then in an initial state until a worker of the factory decides to update the resource and set it into a production state (e.g. by using a PUT request to the address *http://example.org/products/*{*id*}). From now on the customer must be capable to view the state of the production but must not change it. The product state must only be changeable by the workers in the factory.

But the customer is capable to add additional parts to the product and change an individual part's details until a worker closes the part list (by updating the part list state). The customer can add new parts by sending a POST request to the address *http://example.org/products/*{*id*}*/parts*. Parts can only be added by the customer as long as the *http://example.org/products/*{*id*}*/parts* resource is in the *Initial* state. The customer can update dedicated parts by sending a PUT request to the address *http://example.org/products/*{*id*}*/parts/*{*id*}. Individual parts can only be changed as long as they are in the *Initial* state.

Table 1 shows the sample Product API that is used throughout this work as an example. The table lists the aforementioned resources, the possible access methods and the subjects which are allowed to perform the individual API calls.

From the scenario description one can see that the different access methods are bound either exclusively

Table 1: Product API.

	Resource	**Met.**	**Subjects**
1	/products	POST	Customer
2	/products/{id}	GET	Customer, Worker
3	/products/{id}	PUT	Worker
4	/products/{id}/parts	GET	Customer, Worker
5	/products/{id}/parts	POST	Customer
6	/products/{id}/parts	PUT	Worker
7	/products/{id}/parts/{id}	GET	Customer, Worker
8	/products/{id}/parts/{id}	PUT	Customer, Worker

to the customer, exclusively to the workers or to both of them. In addition, the access privileges of some resources depend on an attribute of the resource (the actual state name). The update operation for the product (operation 3), the creation operation of new parts (operation 5), the update operation of the part list (operation 6) and the update operation of individual parts (operation 8) depend on the actual state name.

3 FOUNDATIONS

3.1 Representational State Transfer

Representational State Transfer (REST) is an architectural paradigm that defines constraints on how to scale distributed systems. It is neither a technology nor a standard but a collection of design guidelines. REST was first introduced by Fielding (Fielding, 2000). HTTP (Internet Engineering Task Force (IETF), 1999) is a protocol that supports the design guidelines and that is very often associated with REST. Note that HTTP is a well known example, but there are several other protocols that enable REST. The Richardson maturity model as described in (Webber et al., 2010) defines layers that identify at which level a system supports the different REST constraints.

Level 0: The system is distributed and invokes remote procedure calls. These might be some sort of reusable methods that offer specific services.

Level 1: Resource orientation is likely the most fundamental design guideline for REST. Instead of invoking reusable services, resources are targeted individually. Therefore each resources has a unique address. At this level, access methods are usually encapsulated in the address.

Level 2: HTTP Verbs determine the action that is

performed on resources instead of encapsulating the method into the resources address. The resource address only consists of nouns and the underlying protocol carries the action. Using HTTP, the access method is determined by one of the verbs like *GET*, *POST*, *PUT* or *DELETE*.

Level 3: Hypermedia as the engine of application state (HATEOAS) is clearly the most difficult constraint to understand in theory. But every ordinary web user applies HATEOAS in practice (Richardson, 2010). Web browsers are based on HATEOAS. When using a web browser, the user runs an algorithm:

1) Retrieve a hypermedia representation of a resource.
2) Interpret the representation to get the current resource state.
3) Decide which hypermedia link or form will bring you closer to your goal and click it.
4) Repeat the steps until you got the resource of your liking.

Level 3 of the Richardson maturity model means that the system applies the HATEOAS constraint. That means, a server sends any possible state transition together with the resource to the client. The state transitions are transferred as hypermedia, hence the term hypermedia as the engine of application state.

The payoff of the HATEOAS paradigm is scalability in distributed systems and the decoupling of clients and servers. The goal of scalability can be achieved by a stateless communication between clients and servers, because if the communication is stateless, a server must not hold any session information but can handle every request individually. HATEOAS fulfills this condition, because the server sends the possible state transitions as hypermedia and redirects the management of the application state to the client. Besides the scalability benefit this also means that standardized hypermedia clients can consume the services and no application specific client is required. Therefore, clients and servers are decoupled and a client can automatically adapt to changes on the server side (Richardson and Amundsen, 2013).

3.2 Hypermedia Navigation Model

The approach presented in this paper inherently requires knowledge about the structure of a REST API, particularly about the available resources and the navigation relations between them. There are several metamodels for REST APIs available, including industry-driven languages like Swagger[1], OpenAPI[2] or RAML[3], and also several works that originate from academia (Schreier, 2011; Laitkorpi et al., 2009; Sanchez and de Mattos Fortes, 2014; Haupt et al., 2017). Although HATEOAS is one central feature of REST APIs, most metamodels do not cover it, meaning that they do not provide any means to model hypermedia-based (navigation or other) relations between resources. However, this deficit has been addressed by the metamodel by (Haupt et al., 2014; Haupt et al., 2015) (which is reused in this work), and also lately by the OpenAPI specification v3.0 that introduces the *Link* construct for describing navigation relationships between resources.

The hypermedia-aware metamodel as we described it so far comprises all possible navigation relations between resources. However, at any specific point in time, it might not be possible or allowed to follow all these navigation relations (as illustrated by our running example). There exist two basic approaches that show how this issue can be reflected in the metamodel. The first approach extends the resource model with the capability to model the internal state of resources as well as the relation between this internal state and the supported operations and hypermedia links (Schreier, 2011). The second approach that our work is based on does not focus on describing the internal state of the resources but aims at checking when hypermedia links between resources are active (i.e. shown to the user) and when not. This decision may be based on some resource state, but it may also depend on completely different parameters like e.g. time, the access rights of the user, or its geographical location. In the following sections we will demonstrate that Attribute Based Access Control (ABAC) is a suitable model to describe and evaluate all these conditions and that the combination of a hypermedia-aware resources model with ABAC provides a powerful solution approach for the systematic realization of hypermedia-driven REST APIs.

In section 2 a REST API (cf. Table 1) and the state chart of single resources (cf. Figure 1) in our example scenario are shown. One can see that there is no information about the relations between the resources. The API and the resource state chart do not cover the information if or how the product resource is linked with the part resources. A metamodel that covers HATEOAS addresses this problem.

Figure 2 shows the relations of the product resource from the example scenario. A POST request to the product list creates a dedicated product resource. From Figure 1 one can see that the state of the product can be updated using a PUT request. The metamodel includes the information that this PUT request leads

[1] https://swagger.io/

[2] https://www.openapis.org/

[3] https://raml.org/

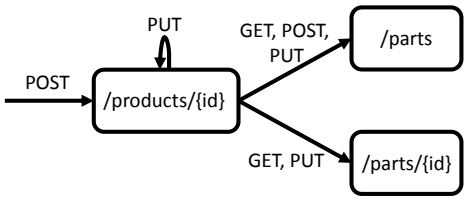

Figure 2: Hypermedia Navigation Model.

the client to the updated product resource. In addition, the metamodel shows that there are state transitions to the part list as well as to dedicated parts. While the state transition to the part list can be passed using a GET, a POST or a PUT request, the state transition to an individual part can be passed using a GET or a PUT request.

For example, a customer may have created a product (*/product/1*) and added two parts (*/product/1/parts/1* and */product/1/parts/2*). Table 2 lists the hypermedia relations of */product/1*, as they are provided by the metamodel. According to Figure 2 the product resource has a reference to itself that can be traversed with a PUT request. The part list (the */parts* subresource of */product/1*) can be fetched with GET, POST or PUT request. Finally, both parts (*/parts/1* and */parts/2*) can be directly consumed following a hypermedia reference in the representation of the */product/1* resource

Table 2: Metamodel example for /product/1.

Resource	Methods
/products/1	PUT
/products/1/parts	GET, POST, PUT
/products/1/parts/1	GET, PUT
/products/1/parts/2	GET, PUT

3.3 Attribute based Access Control

Attribute Based Access Control (ABAC) is an access control model that determines access decision based on attributes. The access control policies that can be implemented with ABAC are only limited by the computational language and the richness of the attributes (Ferraiolo et al., 2015). An attribute can be any property of an entity. For example, a *subject* might have a *name* or a *resource* might have an *URI*. Access then can be restricted (either permitted or denied) depending on whether dedicated attribute values are given. For example, access might only be *granted if the subject name is equal to X and the resource URI is equal to Y*. ABAC enables the specification of rich access policies when compared to classical access control models like RBAC or DAC. It will likely become the dominant access control model in the future because its benefits are too compelling (Sandhu, 2012).

3.4 RestACL

The REST Access Control Language (RestACL) is an access control language and mechanism that is founded on the principles of the ABAC model and that has been designed to fit the requirements of RESTful services (Hüffmeyer and Schreier, 2016c). Therefore, security policies are aligned along resource structures and access is determined on various attributes of different entities, e.g. subjects, resources, actions or other contextual information.

The two major data structures in a RestACL system are domains and policy repositories. While domains are used to quickly map from requested resources to applicable policies, the policy repositories contain all the access policies that are build upon attributes.

Listing 1 shows a domain. It has an entry for the */products/1* resource at the host *http://example.org*. If a GET request is send to that resource, policy *P1* is evaluated. A second entry regulates access to one of the subresources of */product/1*. If a POST request is send to */products/1/parts*, policy *P2* must be evaluated. Note that nested resource entries expand the path attribute and employ the same host as their parent resource.

```
{
  "host": "http://example.org",
  "path": "/products/1",
  "access": [
    {"methods": ["GET"],
     "policies": ["P1"]}
  ],
  "resources": [
    {
      "path": "/parts",
      "access": [
        {"methods": ["POST"],
         "policies": ["P2"]}
      ]
    }
  ]
}
```

Listing 1: RestACL domain.

Listing 2 shows this policy as it is stored in a policy repository. The policy declares two conditions which are logically conjuncted. It becomes applicable in case that the conditions *subject type is equal to Customer* and *resource state is equal to Initial* both are fulfilled. If the policy is applicable, access is permitted.

```
{
  "policies": [
    {
      "id": "P1",
      "description": "Policy example",
      "effect": " Permit ",
      "priority": "1" ,
      "compositeCondition": {
        "operation": "AND",
        "conditions": [
          {
            "function": "equal",
            "arguments": [
              {"category": "subject",
               "designator": "type"},
              {"value": "Customer"}
            ]
          },{
            "function": "equal",
            "arguments": [
              {"category": "resource",
               "designator": "state"},
              {"value": "Initial"}
            ]
          }
        ]
      }
    }
  ]
}
```

Listing 2: RestACL policy.

The RestACL language has been designed to easily integrate with RESTful services and to ensure high-performance access control for those services (Hüffmeyer and Schreier, 2016a). A detailed description of the language can be found in (Hüffmeyer and Schreier, 2016c).

4 AUTHORIZATION-AWARE HATEOAS

4.1 Customized Resources Representations

The HATEOAS constraint requires that a requesting subject must receive all possible state transitions from the server. For example, if the product resource is requested, the resource representation must contain the identical hypermedia controls as they are described by the hypermedia navigation model. Access control in general is not considered in the HATEOAS constraint. That means, if a customer requests the product resource, the response contains the option to update the product even if the customer is never allowed to do so. In contrast, if one of the workers of the factory requests the product resource, the response contains the option to add new parts to the product. But adding new parts is an operation that a workers must never execute.

This problem can be addressed by performing an evaluation of access privileges before the response is send to the client. The operations that must never be executed by the client can be omitted from the resource representation so that the client does not recognize these operations as possible next state transitions. That means, if access privileges are considered in advance, the resource representation is customized to the requesting subject. The customized response only contains those state transitions that the subject is actually allowed to perform. This helps to increase security, avoid unnecessary network traffic and reduce frustration for human web users.

4.2 2-Phase Authorization

In order to customize resource representations in a authorization-aware fashion, two phases of authorization are required to process the resource request. In the first phase the permission to execute the actual resource request is evaluated. In the second phase the permissions to execute the next state transitions need to be evaluated. Figure 3 gives an overview about the customization process.

The process starts with an initial resource request from a subject, e.g. a customer, a worker from the factory or a machine. The request arrives at the Resource Server which immediately forwards it to its Authorization Manager. The Authorization Manager performs a check whether the initial request can be performed by the requesting subject. Therefore, the Authorization Manager sends an access request to a RestACL System. The RestACL System checks whether the initial request is permitted and returns the result back to the Authorization Manager. If the system denies the request, the Authorization Manager returns a message that indicates that the subject is not allowed to perform the request. The Resource Server has to enforce the access decision (e.g. by sending an *HTTP 403 Forbidden* response). If the RestACL System permits the request, the Authorization Manager starts with the computation of the authorization-aware response. Therefore, the Authorization Manager first needs a list of possible state transitions that are linked to the requested resources. Therefore, the Authorization Manager consults the Hypermedia Navigation Model. The Navigation Model knows the possible state transitions for any resource. For example, for the product from the example, the Navigation Model knows that there are state transitions to update

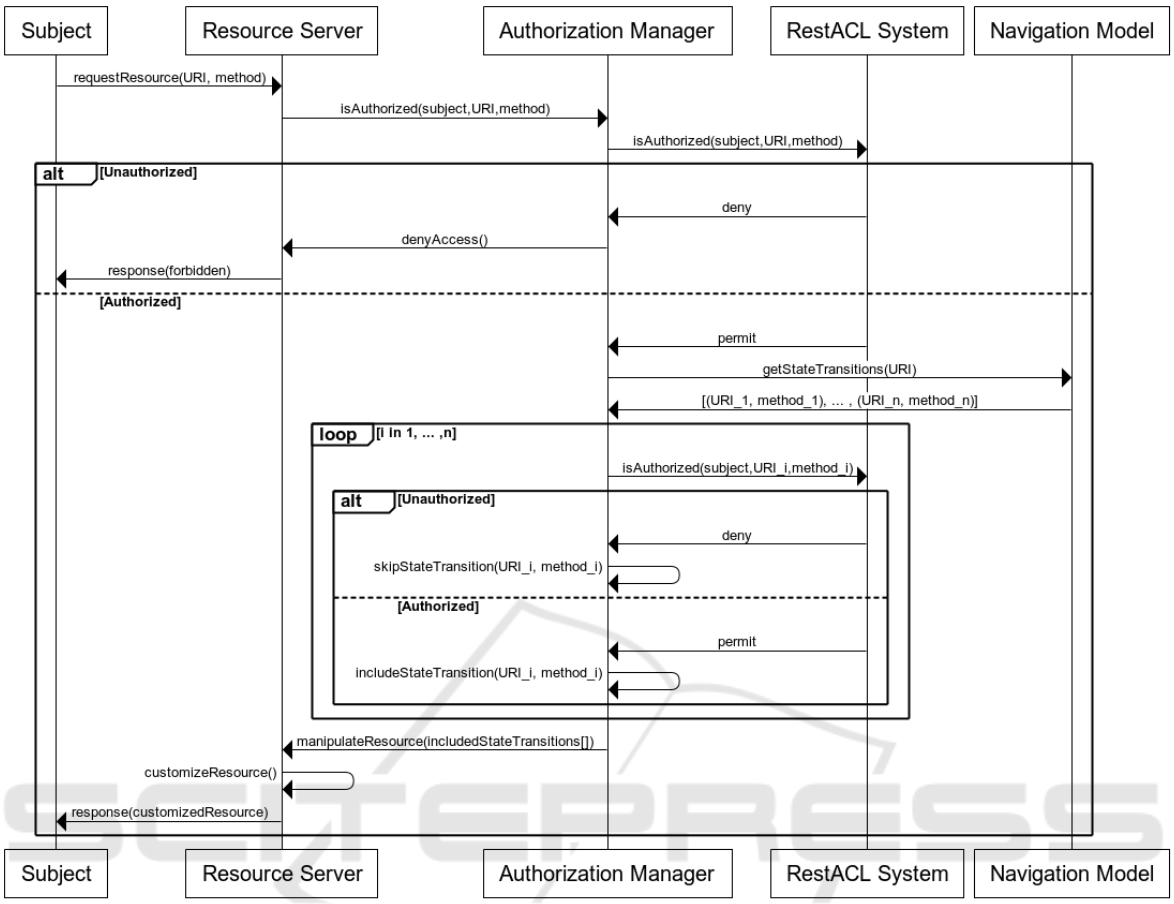

Figure 3: Authorization-aware HATEOAS overview.

the product itself and that there are state transitions to add new parts to the product or view and update existing parts. The Navigation Model returns the given state transitions (as pairs of URIs and methods) for a requested resource to the Authorization Manager. The Authorization Manager then iterates over the list of state transitions. For each state transition an access request is send to the RestACL System which checks whether the requesting subject may perform the state transition. If the subject is permitted to perform the state transition, the state transition is kept within the resource representation. If the subject is not permitted to perform the state transition, the transition is not included from the resource representation. Once all state transitions are either included or skipped, the Authorization Manager responds to the initial request and returns the customized resource back to the Resource Server. The Resource Server then needs to adjust the response according to included state transitions and send it to the Subject.

While Subject, Resource Server and an access control system are well known components in a RESTful architecture, the Authorization Manager and the Hypermedia Navigation Model are new components. The Authorization Manager is a coordinator of the access control operations. It coordinates the process of building an authorization-aware response. That means, the Authorization Manager hides all access control operations to the Resource Server. When a request arrives, it consults the Navigation Model and creates a set of access request from the set of possible state transitions as provided by the Navigation Model. These access requests are actually evaluated by the access control system.

One can see there are two phases of authorization. In Phase 1 the execution of the initial request is either permitted or denied. In Phase 2 access to the related resources is evaluated. The differentiation between the two phases is important since in both phases there are different attribute types that need to be evaluated.

4.3 Authorization Process Example

Before we discuss different attribute types in Section

4.4, we take a closer look at the product example. In the real world, there might be several access policies that permit or deny writing access to the product resource. But to explain the concept of authorization-aware HATEOAS, we focus on only two policies.

The first policy grants access to the customer. *P1* (cf. Listing 3) grants access if the requesting *subject* is of the *type Customer* and the requested *resource* is in the *Initial state*.

```
{
  "policies": [
    {
      "id": "P1",
      "description": "Customer access policy",
      "effect": " Permit ",
      "priority": "1" ,
      "compositeCondition": {
        "operation": "AND",
        "conditions": [
          {
            "function": "equal",
            "arguments": [
              {"category": "subject",
               "designator": "type"},
              {"value": "Customer"}
            ]
          },{
            "function": "equal",
            "arguments": [
              {"category": "resource",
               "designator": "state"},
              {"value": "Initial"}
            ]
          }
        ]
      }
    }
  ]
}
```

Listing 3: Customer access policy.

The second policy from the Industry 4.0 scenario restricts worker access. *P2* (cf. Listing 4) grants access if the requesting *subject* is of the *type Worker* and the *resource* is not in the *Completed state*. This ensures that a worker can not update resources anymore once the production process is finished.

According to the scenario description in section 2 these two policies must be assigned to the resources as indicated in Listing 5 to control access to resources as described in the example scenario.

Both - customer and workers - can perform a reading access (by sending a *GET* request) to the product resource, the part list and to the individual parts at any time. Therefore, access to these resources must not be restricted. But only the workers of the factory can update the state of the product (by sending a *PUT* request to the */product/1* resource), while only the customer can add new parts of the product (by sending a *POST* request to the */parts* subresource of the product as long as the */parts* resource is in the *Initial* state).

```
{
  "policies": [
        {
      "id": "P2",
      "description": "Worker access policy",
      "effect": " Permit ",
      "priority": "1" ,
      "compositeCondition": {
        "operation": "AND",
        "conditions": [
          {
            "function": "equal",
            "arguments": [
              {"category": "subject",
               "designator": "type"},
              {"value": "Worker"}
            ]
          },{
            "function": "unequal",
            "arguments": [
              {"category": "resource",
               "designator": "state"},
              {"value": "Completed"}
            ]
          }
        ]
      }
    }
  ]
}
```

Listing 4: Worker access policy.

```
{
  "host": "http://example.org"
  "path":"/products/1",
  "access":[
    {"methods":["PUT"],"policies":["P2"]}
  ],
  "resources":[
    {
      "path":"/parts",
      "access":[
        {"methods":["POST"],"policies":["P1"]},
        {"methods":["PUT"],"policies":["P2"]}
      ],
      "resources":[
        {
          "path":"/parts/{id}",
          "access":[
            {"methods":["PUT"],
             "policies":["P1, P2"]}
          ]
        }
      ]
    }
  ]
}
```

Listing 5: Resource-policy assignment (domain).

If the customer requests the product (identified by the path */products/1*), policy *P1* is evaluated in the first phase. The policy grants access and the Authorization Manager consults the Hypermedia Navigation Model which returns all possible state transitions. For every state transition the Authorization Manager creates an access request and passes it to the RestACL System. The system computes whether the subject is allowed to perform the state transition or not. For example, the customer is not allowed to update the product resource itself. A PUT request is only permitted to the workers of the factory. Therefore, the Authorization Manager excludes this state transition. But the customer can add new parts by using POST requests. This state transition is kept and send to the customer in the final response.

4.4 Static and Dynamic Attributes

One important aspect for an authorization-aware HATEOAS system is the time factor. In RESTful applications that apply HATEOAS, the communication between client and server is stateless and client and server are decoupled. A client requests a resource and the server responds with a representation of the resource including several state transitions. The presence of the state transitions does not specify any restrictions about *when* the state transition can or must be performed. This can happen immediately, within an hour, after a day, a week or even a month. In consequence, it is not possible to compute access decisions for all state transitions at the time the resource is requested. The attribute values possibly change until the next request arrives. Therefore, no reliable prediction can be made, if the attribute value can change. That means, if attribute based access control is used to compute access decisions, one needs to distinguish between static and dynamic attributes.

Definition (Static/Dynamic Attribute): An attribute is dynamic if the attribute value might change between two requests from the same subject. An attribute is static if the attribute value does not change until the next request arrives.

That means, an attribute is called static, if it is guaranteed that there is no change to the attribute value until the next state transition is done. A potential future state transition can be executed at any time and the resulting access decision can not always be precomputed because the values of dynamic attributes might change. If the access decision relies on dynamic attributes, an undetermined access decision is send from the RestACL system to the Authorization Manager. The Authorization Manager includes the related resource references as long as there are no static attributes that prevent the applicability of the access policy. Attribute conditions targeting dynamic attributes are only evaluated if an resource request is actually performed.

A look at the product example shows the difference between static and dynamic attributes. Considering that the subject has a type like customer or worker, this attribute will not change between two requests and therefore can be declared static. On the other side the resource state (e.g. *Initial* or *In Production*) might change between two requests. Therefore, an attribute *state* must be declared dynamic. The distinction between static and dynamic is application dependent and is an additional task of the access control design.

4.5 Authorization-aware Responses

In (Liskin et al., 2011) the authors describe how HATEOAS support can be added to existing services that do not support all of the REST constraints. The main idea is to add *Link* headers in HTTP responses to include the HATEOAS support. Every state transition is added a a *Link* header. Therefore, they introduce a wrapper between client and server. The wrapper captures the response from the HTTP server and performs a look-up for possible state transitions. The transitions are stored in a so called state chart. Knowing the possible transitions, the wrapper can create and insert hyperlinks to the next states.

We extended this approach and included the execution of access control logic. Instead of simply adding all theoretically possible state transitions to the response, an access control request is performed for every state transition and they are only added if access is not prohibited. As described in the previous sections, those additional access requests only consider attribute conditions with static attributes. Attribute conditions based on dynamic attribute values are interpreted as applicable. If the access control system allows the state transition (or it might be allowed in the future), the Authorization Manager adds a header to the response. In case that static attributes prevent the applicability, the header is not included in the response.

In the product scenario the customer or the workers of the factory might request the product resource using a *GET* request as indicated in Listing 6.

```
GET /products/1 HTTP/1.1
Host: example.org
```

Listing 6: HTTP request.

If no authorization-aware HATEOAS is performed the server would return a response including

any known state transition (cf. Table 2), no matter whether the requesting subject is allowed to perform it or not. All subjects receive the same response (cf. Listing 7) and resource representation, resulting in subjects that might try to access the resource.

```
HTTP/1.1 200 OK
Link: </products/1>; verb="Put"
Link: </products/1/parts>; verb="Get,Post,Put"
Link: </products/1/parts/1>; verb="Get,Put"
```

Listing 7: Non-authorization-aware response.

If authorization-aware HATEOAS is done, the resource representation can be customized to the requesting subject depending on the resource state or any other attribute. If the customer requests the product resource, the state transition to update the product itself is not included, because the customer is not allowed to do so at any time (cf. Listing 8). The same applies for the state transition to update the part list. Note that the *Put* operation is not included in the *Link* header.

```
HTTP/1.1 200 OK
Link: </products/1/parts>; verb="Get,Post"
Link: </products/1/parts/1>; verb="Get,Put"
```

Listing 8: Customer customized response.

If authorization-aware HATEOAS is done and the workers of the factory request the product resource, the state transition to update the product is included, but the state transition to add new parts of the product is skipped (cf. Listing 9). Note that the worker of the factory can only perform *Get* or *Put* requests on the part list but not *Post* requests. The option to send a *Post* request to the part list is not included in the response.

```
HTTP/1.1 200 OK
Link: </products/1>; verb="Put"
Link: </products/1/parts>; verb="Get,Put"
Link: </products/1/parts/1>; verb="Get,Put"
```

Listing 9: Worker customized response.

Note that we extended the *Link* header and added the verb property. This is a easy way to instruct the client what methods can be used to access the resource.

The use of *Link* headers works fine in machine-2-machine communication. In order to provide an appropriate response to a human user, the Resource Server must be capable to render the response so that only references mentioned in the *Link* headers are displayed. Note that details about rendering of representations are out of the scope of this paper but are part of our future work.

5 EVALUATION

The delay that is caused by the additional authorization checks might be a critical number. If the efforts are too high, the authorization system might become a bottleneck that slows an entire application down. Therefore, it is crucial to analyze how the delay is composed and measure its size in practice. The overall processing time t to access a resource in an authorization-aware fashion can be computed as the sum of the time t_1 to process the resource request and the time t_2 to build the authorization-aware response:

$$t = t_1 + t_2 \tag{1}$$

$$t_1 = t_{network} + t_{resource} + t_{access} \tag{2}$$

$$t_2 = t_{model} + n * (t_{access} + t_{manipulation}) \tag{3}$$

The time intervals are listed in Table 3 and n is the number of possible next state transitions as given by the Hypermedia Navigation Model.

The first phase of the 2-phase authorization process (cf. Section 4.2) is represented in t_1 as t_{access}, while t_2 describes the second phase that enables authorization-aware processing.

Table 3: Processing times.

t_network	The network runtime (request and response) between the requesting subject and the resource server and the processing time for the underlying protocol stack (e.g. IP, TCP, HTTP).
t_resource	The time that is required to send an access request from the resource server to the authorization manager and enforce the decision.
t_access	The time that is required to send an access request from the authorization manager to the access control system and receive a response.
t_model	The time that is required to send a request from the authorization manager to the model service and to receive the state list as response.
t_manipulation	The time that is required to manipulate the response (include or skip a state transition).

To verify the functionality and measure the performance of the authorization-aware processing, we

created a test setup. In this setup the Resource Server and the Authorization Manager have been integrated into a single software component. Additionally, the RestACL system and the Hypermedia Navigation Model have been set up on the same machine as the Resource Server/Authorization Manager component. To minimize the effects of the network runtime, also the requesting client ran on the same machine. The tests have been executed on a quad core unit with 2.90 GHz clock speed. 2 GB memory have been reserved for testing purposes only.

We did an analysis on the RESTful services of the example scenario. That means, we created a RESTful API as described in section 2 using JAX-RS[4] and a metamodel as described in section 3.2. The RestACL system has been setup to protect a product with two parts. Note that the RestACL system has been analyzed in detail in (Hüffmeyer and Schreier, 2016a; Hüffmeyer and Schreier, 2016b). It was demonstrated that the processing time for an access request is independent from the number of resources that are addressed in a domain description. Therefore, already a small amount of resources is sufficient to test the performance of the authorization-aware system.

For testing purposes the product, the part list and the individual parts have been requested. For each resource we created requests that cover different user types (*Customer* or *Worker*) and different resource states (*Initial*, *In Production* or *Completed*). Each request has been executed at least ten times.

Table 4 lists the min., max. and average processing times (in milliseconds) that we observed for an authorization-aware HATEOAS request. As one can see, the processing times are very small and differ only slightly. The biggest interval between min. and max. processing time is given for an access control request. But even this distance is less than 2ms.

Table 4: Test results.

	t_resource	t_access	t_model	t_manipulation
min.	0.002	0.270	0.344	0.001
max.	0.015	2.073	0.792	0.031
avg.	0.003	0.532	0.445	0.009

According to (3), the average additional delay for a resource with 6 state transitions (like the example from Listing 7) can be computed to:

$$t_2 = 0.445ms + 6 * (0.532ms + 0.009ms) = 3.691ms$$

About 3-4ms are small compared to the average network runtime between a client and a resource server even in a local network. For example, t has been measured as about 20ms in the test setup. Therefore, the additional delay caused by the authorization-aware processing is negligible in the test setup for the example scenario. This might change if the resource has much more than 6 state transitions, because the delay heavily depends on the number of possible state transitions for a resource. Also the manipulation time may vary much more if the actual resource representation must be adjusted. In a machine-2-machine scenario the inclusion and exclusion of *Link* headers works fine, but if the resource representation must be tailored for humans, higher computation efforts can be expected.

[4]https://jcp.org/en/jsr/detail?id=311

6 RELATED WORK

Authorization in the context of REST has been discussed in detail when it comes to sharing resources in the Internet. OAuth is a protocol that enables the user to share its data in various applications (Internet Engineering Task Force (IETF), 2012). For example, if the user has two accounts on different social media platforms, the user can grant the first application to access his data in the second application without revealing the users credentials in the first application to the second application. An approach to centralize permission management using OAuth tokens is described in (Memeti et al., 2015). The authors present an approach in which a centralized coordinator issues tokens to clients which enable the clients to access different services.

User Managed Access (UMA) is another protocol that targets sharing of resources (Internet Engineering Task Force (IETF), 2015). While OAuth focuses on resource sharing between applications, UMA enables the user to grant access privileges to other users instead of applications. UMA defines message sequences that describe how to manage access permissions and how to retrieve access tokens. Like OAuth, UMA does not specify any details about access request evaluation or permission storage.

In (Bhatti et al., 2005) the authors describe that a key challenge in securing Web Services is the design of effective access control schemes. The authors try to solve this problem using a context-aware approach to secure access. The approach extends Role Based Access Control (RBAC) and includes context information (e.g. time and location) to determine access privileges. In (Jin et al., 2012) the authors show that ABAC is capable to cover multiple Access Control Models like Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC). Therefore, ABAC can be

seen as an appropriate model that covers many of the challenges of access control for Web Services.

Authorization-aware representations of resources is clearly a topic that has been very rarely discussed in science. Adaptive Hypertext and Hypermedia is an research area that targets customized representations of resources depending on user knowledge or preferences (Brusilovsky, 1998). Approaches from that research area try to increase user experience by delivering content according to what the user wants to see (Höök et al., 1995; Kaplan et al., 1993) rather than enforcing access rights. Therefore, such approaches are focused on the user and the modeling of the user but do not target the evaluation of any kind of attributes. Authorization awareness is also an approach to increase the user experience, but the way to achieve this goal is very different.

7 CONCLUSION AND FUTURE WORK

This works extends the HATEOAS principle of REST and introduces authorization awareness for RESTful services. Therefore, in an additional authorization phase all state transitions that would lead the requesting subject to another application state, but that are not executable for the subject, are skipped. This leads to customized resources depending on what subject requests the resource. We have proven the functionality of the approach in an example scenario from the Industry 4.0 area. The implemented solution adds a negligible delay to the overall processing time for the resource request.

In our future work we want to perform a more detailed analysis on large amounts of resources, since scalability is one of the major benefits of REST and the approach must be capable to handle large amounts of data. It has been already shown that the RestACL system can easily handle large amounts of data (Hüffmeyer and Schreier, 2016a). Therefore, we expect the authorization-aware HATEOAS approach to be scalable in practice, too.

As we have mentioned previously, we implemented the customized resources using *Link* headers. This works totally fine in machine-2-machine communication. But if the requesting subject is a human using a web browser, one needs additional renderers that exclude the skipped state transitions from the body part of the response. Therefore, a manipulation of different content types must be performed. The design and implementation of such manipulators is also be part of our future work.

REFERENCES

Amundsen, M. (2017). *RESTful Web Clients - Enabling Reuse Through Hypermedia*. O'Reilly Media.

Bhatti, R., Bertino, E., and Ghafoor, A. (2005). A Trust-Based Context-Aware Access Control Model for Web-Services. In *Distributed and Parallel Databases, Vol. 18*. Springer.

Brusilovsky, P. (1998). Methods and techniques of adaptive hypermedia. In *Adaptive Hypertext and Hypermedia*. Springer.

Ferraiolo, D., Kuhn, R., and Hu, V. (2015). Attribute-Based Access Control. In *Computer, Vol.48*. IEEE Computer Society.

Fielding, T. R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine.

Haupt, F., Karastoyanova, D., Leymann, F., and Schroth, B. (2014). A Model-Driven Approach for REST Compliant Services. In *ICWS '14 - 2014 IEEE International Conference on Web Services*. IEEE.

Haupt, F., Leymann, F., and Pautasso, C. (2015). A conversation based approach for modeling REST APIs. In *WICSA '15 - 12th Working IEEE / IFIP Conference on Software Architecture*. IEEE.

Haupt, F., Leymann, F., Scherer, A., and Vukojevic-Haupt, K. (2017). A Framework for the Structural Analysis of REST APIs. In *ICSA '17 - Proceedings of the IEEE International Conference on Software Architecture*. IEEE.

Höök, K., Karlgren, J., Wærn, A., Dahlbäck, N., Jansson, C. G., Karlgren, K., and Lemaire, B. (1995). A Glass Box Approach to Adaptive Hypermedia. In *Adaptive Hypertext and Hypermedia*. Springer.

Hüffmeyer, M. and Schreier, U. (2016a). Analysis of an Access Control System for RESTful Services. *ICWE '16 - Proceedings of the 16th International Conference on Web Engineering*.

Hüffmeyer, M. and Schreier, U. (2016b). Formal Comparison of an Attribute Based Access Control Language for RESTful Services with XACML. *SACMAT '16 - Proceedings of the 21st ACM Symposium on Access Control Models and Technologies*.

Hüffmeyer, M. and Schreier, U. (2016c). RestACL - An Attribute Based Access Control Language for RESTful Services. *ABAC '16 - Proceedings of the 1st Workshop on Attribute Based Access Control*.

Internet Engineering Task Force (IETF) (1999). Request for Comments: 2616 - Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616*.

Internet Engineering Task Force (IETF) (2012). Request for Comments: 6749 - The OAuth 2.0 Authorization Framework.

Internet Engineering Task Force (IETF) (2015). Internet-Draft: User-Managed Access (UMA) Core Protocol.

Jin, X., Krishnan, R., and Sandhu, R. (2012). A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. *DBSec '12 - Proceedings of the 26th Annual Conference on Data and Applications Security and Privacy*.

Kaplan, C., Fenwick, J., and Chen, J. (1993). Adaptive hypertext navigation based on user goals and context. In *User Modeling and User-Adapted Interaction*. Springer.

Laitkorpi, M., Selonen, P., and Systa, T. (2009). Towards a model-driven process for designing restful web services. *ICWS '09 - Proceedings of the 2009 IEEE International Conference on Web Services*.

Lasi, H., Fettke, P., Kemper, H.-G., and Feld, T. (2014). Industry 4.0. In *Business & Information Systems Engineering*. Springer.

Liskin, O., Singer, L., and Schneider, K. (2011). Teaching Old Services New Tricks: Adding HATEOAS Support as an Afterthought. *WS-REST '11 - Proceedings of the Second International Workshop on RESTful Design*.

Memeti, A., Selimi, B., Besimi, A., and Cico, B. (2015). A Framework for Flexible REST Services: Decoupling Authorization for Reduced Service Dependency. *MECO'15 - Proceedings 4th Mediterranean Conference on Embedded Computing*.

Richardson, L. (2010). Developers Like Hypermedia, But They Don't Like Web Browsers. *WS-REST '10 - Proceedings of the First International Workshop on RESTful Design*.

Richardson, L. and Amundsen, M. (2013). *RESTful Web APIs - Services for a Changing World*. O'Reilly Media.

Sanchez, Robson Vincius Vieira, R. R. d. O. and de Mattos Fortes, R. P. (2014). RestML: Modeling RESTful Web Services. In *REST: Advanced Research Topics and Practical Applications*. Springer.

Sandhu, R. (2012). The authorization leap from rights to attributes: maturation or chaos? *SACMAT '12 - Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*.

Schreier, S. (2011). Modeling restful applications. In *Proceedings of the Second International Workshop on RESTful Design, WS-REST 2011, Hyderabad, India, March 28, 2011*.

Webber, J., Parastatidis, S., and Robinson, I. (2010). *REST in Practice - Hypermedia and Systems Architecture*. O'Reilly Media.

Wilde, E. and Pautasso, C. (2011). *REST: From Research to Practice*. Springer.