Hecate,
Managing Authorization with RESTful XML

Sebastian Graf
University of Konstanz
Departement of Computer and
Information Science
Konstanz, Germany
sebastian.graf@
uni-konstanz.de

Vyacheslav Zholudev
Jacobs University Bremen
School of Engineering and
Science
Bremen, Germany
v.zholudev@
jacobs-university.de

Lukas Lewandowski
University of Konstanz
Departement of Computer and
Information Science
Konstanz, Germany
lukas.lewandowski@
uni-konstanz.de

Marcel Waldvogel
University of Konstanz
Departement of Computer and
Information Science
Konstanz, Germany
marcel.waldvogel@
uni-konstanz.de

ABSTRACT

The potentials of *REST* offers new ways for communications between louse coupled entities featured through the *Web of Things* [12]. The binding of the disjunct components of this architecture creates security issues, such as the centralized authorization techniques respecting the independence of the underlying entities. This results in the question how authorization is performed respecting the flexibility of *REST* without any knowledge about the underlying resources. Nevertheless, possible knowledge about these resources should enable the authorization workflow to offer finer-granular permissions on substructures of the resources. With our new approach - we named Hecate- we offer a framework to assure simplified handling while keeping the potentials and flexibility of *REST*. We have designed an architecture based on XML with a flexible authorization mechanism on the one hand and optional resource-awareness on the other hand. The flexibility within the authorization work-flow bases on permission sets respecting the *HTTP*-verbs. Additional in-depth knowledge of the entity optionally extends these permissions with resource-aware filters. Hecate offers not only great benefits because of its flexibility, but also because of the optional extensibility proved within the two reference implementations. With Hecate, we show that a centralized authorization mechanism combining independence and optional resource-based filtering extends the flexibility of *REST* rather than restricting it.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

Keywords

XML, REST, Authorization

General Terms

Design, Management, Security

1. INTRODUCTION

1.1 REST and its flexible usage

REST [9] represents a flexible yet powerful technique to support communication in distributed environments. *HTTP* as its base evolves from a communication protocol to an application protocol based on the accretive interaction of services in heterogenous environments. One existing example is the *Web of Things* [12]: Variable items interact in this paradigm with each other whereas *HTTP* acts as the communication- and application-protocol providing direct stateless access and operations. The reason for using *REST* as interaction technique lies in the supported operations going along with the *HTTP*-verbs, the lack of complex handling of states and the variability regarding resource characteristics. Based on this flexibility, *REST* itself is able to work with all kinds of resources even if they are often represented [16] by customizable formats like XML [3] or JSON [5].

However, the direct allocation of items over common interfaces like *HTTP* motivates security questions e.g. how to guard the access on these resources in a flexible and scalable manner. One solution lies in the centralization of the authorization on commonly provided resources. Such a centralized permission model offers great benefits against resource-based authorizations e.g. security consistency within all provided services and reduced overhead regarding shared rules. Those rules would be applicable on different resources and
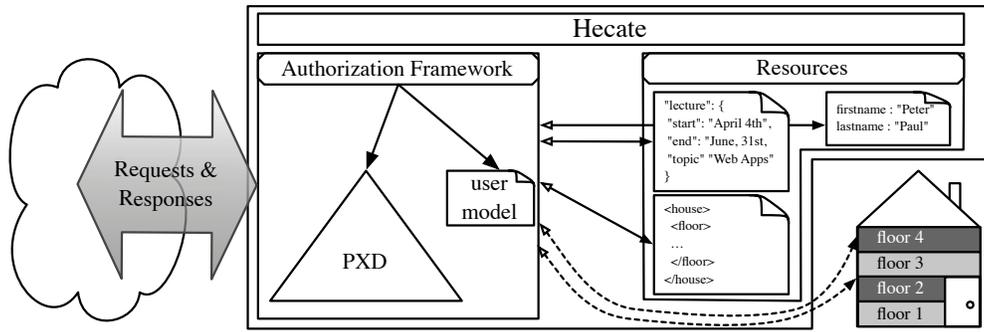
Figure 1: Overview of proposed architecture

result in an integrative and transparent authorization architecture applied to a flexible resource handling which is based on *REST*.

1.2 Problem Statement and Contribution

Even if *HTTP* offers easy yet entrenched techniques for authenticating users, a centralized authorization endangers the independence of the underlying resources. Authorization relying on in-depth knowledge regarding the characteristics of these entities would restrict the functionality of *REST*-based environments. Such necessary awareness acts as a contradiction regarding the flexibility and independence of resources which must be preserved within a centralized authorization architecture. Nevertheless, if a centralized authorization mechanism makes benefit from optional knowledge about commonly resources regarding resource-aware permission appliances, this would enable such an approach to offer permission rules not only mappable on the resource but also on defined substructures of this entity. However, this knowledge must influence any authorization workflow in an extensional way to prohibit any restriction regarding the independence of the different resources.

Based on this demanded feature-set, the question arises what issues must be respected by designing an integrative and transparent authorization architecture respecting the flavors of *REST* with optional support of permissions on substructure level. In our approach HECATE[1], we present such an integrative system satisfying the following constraints:

- The variable representations of the underlying resources must be respected. Based on the variety of "things" representing those entities, this point is crucial for not reducing any functionality. That means that knowledge of the underlying resource is beneficial but not mandatory.

- Different permissions should be applied to one resource. Therefore those permissions should refer to the same corresponding URI as well since the operations denoted by the *HTTP*-verbs can be reflected by disjunct permission-sets. Optional knowledge about the resource and applicable modification techniques should be registered within these different permissions.

- The extensibility of *REST* must be kept. This includes easy adaption of our architecture regarding new

---

[1]Hecate is a greco goddess associated with magic and crossroads.

operations, new users and new resources as well as extensions on the operational level of the resource itself (e.g. content types). Each adaption should only result in constant overhead with respect to our authorization architecture.

HECATE consists out of a modular infrastructure acting as a layer handling authorization requests in a centralized manner. This is achieved by surrogating the access to commonly registered resources through an adaptable authorization mechanism. HECATE therefore consists out of a **P**ermission **X**ML **D**ocument called *PXD* for the registered resources within a resource provider and a user model for mapping the specific users to the *PXD*. Since *PXD* is based on XML, it has the ability to either be equipped with links to forward the requests or to store direct content in its structure. The user model maps to the rules defined in the *PXD* which are referenced with the help of fixed defined XPath [4]-expressions. This combination offers us coverage of underlying resources with *HTTP*-operation-based access rules. The number of permissions is not restricted by linking one specific ruleset to one user instead the combination of rules associated to one user takes place in the user model only. Each rule within the *PXD* is furthermore able to support optional resource-aware filtering depending on the requesting *HTTP*-verb as well as on known characteristics of the entity.

Utilizing the extensibility of XML, we provide an easy and straight-forward way to prove any incoming requests against centralized permission-sets. Our implemented architecture consists of a service where the user model and the *PXD* feature the main components. This combination offers a highly extensible and flexible mechanism for satisfying resource-aware authorization needs within single encapsulated requests. The *PXD* provides an easy way to directly integrate content and to optionally intermediate filters resulting in views and even finer-granular permissions on the requested resource. These filters require in-depth knowledge of the characteristics of the entity and enable requests as well as responses to act on an even finer level than the direct resource allocation over an URI.

1.3 Related Work

The *eXtensible Access Control Markup Language(XACML)* [13] represents the base regarding authorization for XML documents. Permissions and roles are reflected by a straight-forward XML dialect. Damiani et al. [6] present a model where permission roles on subtrees are bound to XPath-

expressions. A simple table maps the different permissions to users. This approach is extended: Gabillon [10] makes use of this idea by extending the proposed functionality for updating purposes of native XML databases. Fan et al. [8] defines furthermore views based on a computed DTD of the permission model. We use the idea of Damiani [6] as well since our permission model constitutes out of fixed XPath-expressions. Since we rely on independent resources denoted by links instead of direct XML as database, we further specify our authorization with respect to *REST*. This results in our fixed schema described in Section 2.2.

X-RBAC [2] and its extension *X-GTRBAC* [1] represents a policy specification of XML-based web services. In this approach, rules and permissions find themselves in an XML dialect similar to our approach. Even if this approach relates to web services in general, it differs from HECATE since we rely specifically on *REST* with independent resources and not on XML as common resource format in a service context. Related to *SOAP*, Damiani et al. [7] presents an approach quite similar to ours. This approach encapsulates inlying resources for authorization purposes specifically for *SOAP*-based communication. Current *HTTP* approaches mostly rely on authentication. These approaches e.g. from Story et al. [15] or Peng et al. [14] offer possible extensions to HECATE for authentication purposes.

2. HECATE

Since resources tend to have various different characteristics, HECATE cannot rely on those specific peculiarities since any adaption of the authorization process to concrete underlying formats might result in a restriction of the overall functionality. Nevertheless, in-depth knowledge about the architecture of a resource enables an authorization workflow to offer extended functionality represented by permissions on substructure level of a resource. HECATE satisfies these needs by an indispensable authorization workflow against the URI and an optional, extending authorization workflow against known substructures. More concise, based on the URI, the *HTTP*-verb and the user credentials, the request is either forwarded, denied or equipped with an optional filter which is resource-aware. HECATE acts as a proxy whereas the representation of the underlying resource stays flexible and unbound but can influence the authorization in an extensible way.

Figure 1 shows HECATE at a glance. HECATE includes an authorization framework and a multiple number of resources. The authorization framework consists out of a user model, representing the authorization mapping to the user credentials, and the **P**ermission **X**ML **D**ocument (*PXD*), representing different rules and their mapping to *HTTP*-functionalities, resources and optional resource-aware filters. The resources are either stored directly in HECATE or linked in the *PXD*. Note that each request is evaluated beforehand by the authorization framework before consulting any underlaying resources.

2.1 User Management

The user model acts as a central storage regarding user identifiers and suitable references to the *PXD* over unique identifiers. Table 1 shows an example of the user model.

The user store consists out of the user-ids which are mapped to identifiers of different rules. "john.doe" is allowed to access ruleset 13 while "jane.doe" is permitted to access the

rulesets 12 and 13. As one URI can be accessed through different rulesets as denoted in Sec. 2.2, the user model itself is not aware about the concrete mapping of user-associated rules to requested URIs. Since each rule maps to one specific *HTTP*-operation related to an unique URI, each authorization performing an operation on a defined resource is referenced only once in the user model.

Since we work on *HTTP*-operational level, already existing rules often match the requirements of a new user to be inserted. A new user is simply inserted into the user store including the mapping to the suitable permissions. If there is no matching ruleset available for the denoted URI, a new one is inserted in the *PXD* and referenced within the user model.

As clearly visible, the user model is designed straightforward with less overhead while most logic regarding our permission handling is included in the *PXD*.

2.2 Permission XML Document

The *PXD* represents HECATE related to rules, permissions and additional filtering. The motivation for the architecture of the *PXD* finds itself in the following four aspects:

1. A registered URI can be guarded by multiple rules. Based on the different possible operations on one URI, this aspect ensures a variable number of different authorization sets.

2. Each rule maps on one specific *HTTP* operation derived from the set of available *HTTP*-verbs. Therefore we ensure unique *REST* awareness within each rule in the *PXD*.

3. An additional optional resource-aware permission filtering is provided besides the authorization on URI- and *REST*-level. Even if this filtering is independent from the data itself, it must be aware of the characteristics of the data.

4. The resource can be referenced over links or stored in the *PXD* itself. In both cases, the content related to a resource remains independent against mapping rules and permissions.

A fixed schema shown in Fig. 2 defines the *PXD* where each of the four aspects maps to corresponding nodes including suitable substructures:

- *rule*-nodes:
  Each *rule*-node represents one specific permission. This permission is bound to one specific resource over the *uri*-attribute. Even if this attribute is mandatory, it should not be used for unique identification since multiple rules can map on the same URI allowing different operations. For unique identification of rules, an *id*-attribute is included within each rule linking the user model to the *PXD*. The concrete permission on the

Table 1: User Permissions

user-id	rule-ids
john.doe	13
jane.doe	12 13
. . .	. . .

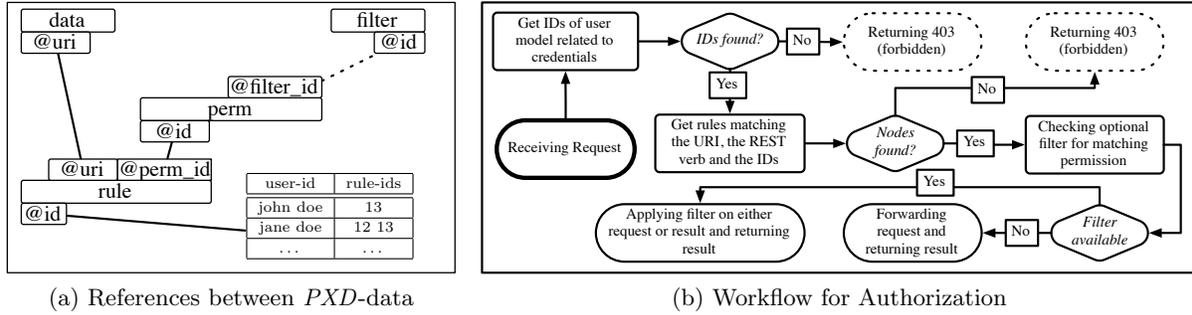(a) References between *PXD*-data   (b) Workflow for Authorization

Figure 3: *PXD*-schema and relations

resource is not stored directly in the *rule*-node but referenced over an extra *perm*-node.

- *perm*-nodes:
  The actual permissions are represented by the *perm*-nodes. Independent from an URI, each permission represents exactly one single *HTTP*-verb and optionally links again a resource-aware filter. The referencing of such a permission takes place over a dedicated *id*-attribute linked within each *rule*-node.

- *filter*-nodes:
  Each filter results in a *filter*-node. With an architecture similar to the *data*-nodes, the *filter*-nodes represent resource-aware filters applied after the mandatory authorization. Since the filters are uniform and therefore combinable with multiple similar resources, we register those not related to the URI. Instead we equip the *filter*-nodes with an own unique identifier denoted as *id*-attribute. This identifier is optional referable within each *perm*-node. Section 2.3 describes the architecture of the resource-aware filtering mechanism in detail.

- *data*-nodes:
  Each *data*-node represents one specific resource consisting either of a link or direct content. The content can be made out of any type representing the flexibility of XML as underlying resource format whereas a *link*-node must contain a link for forwarding the request. The choice of using either content or a link is exclusive and mandatory. The referencing against the concrete resource takes place over an attribute which maps the requested URI. Therefore, each URI is bound

to exactly one *data*-node whereas the same underlying data can be stored or referenced within multiple *data*-nodes. In this *data*-node, the *uri*-attribute acts as a primary key related to the resource.

The *rule*- and the *perm*-nodes represent based on the description above the central instance for permission-handling. All operations are either granted or revoked based on the URI and the related *HTTP*-verb which are checked against the *rule*- and *perm*-nodes. Figure 3a shows the relations to other elements within the *PXD* based on fixed defined *IDREF*s. Since *rule*-nodes and *perm*-nodes rely on the mapping of single *HTTP*-operations to unique URIs, the aggregation of the rules takes place in the user model only. Even if former versions of our *PXD* offered multiple resources referable within one rule, *rule*-nodes now only contain one link to a resource, since the effort of combination of common rules enables HECATE to combine and adapt existing rules in a more flexible way than the usage of complex rules. The same reason resulted in the reference of single *HTTP*-operations against unique permissions instead of the combination of multiple operations mapped on one *rule*-node. Such complex rules tend to degenerate due to the necessity of cross-checks since other referencing users are affected within modifications and adaptions of those rules.

Regarding adaptions, HECATE provides high flexibility: Any new resource ends up in a *data*-node. Further, one or multiple *rule*-nodes represent the permitted operations by linking to suitable *perm*-nodes which are created if not already existing. If necessary, this creation includes an optional resource-aware filter. Due to the independence of the *filter*-, and *perm*-nodes against the URI, both are combinable within common and similar resources in dedicated *rule*-nodes.

Figure 3b shows the workflow of information processing in HECATE within each request. Since each request contains the information about the requesting user and the URI, these two pieces of information are used to identify the rule applied on the request. The identification takes place as XPath-expression. The expression consists out of the requested URI, the *HTTP*-verb and the ids retrieved from the user model. All information is combined via *INTERSECT*s in XPath-predicates which ensures scalability regarding the complexity of the rule-retrieval process. After retrieval of the matching rule, optional filters are dereferenced. If no filter is registered within the matching rule, the request is simply forwarded to the resource. In the case of a filter registration, either the content of the request or of the response
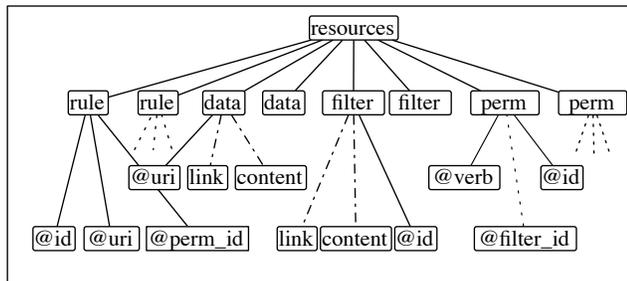

Figure 2: Schema of *PXD*

Figure 4 (a) Data-Aware Response:

GET john.doe:secretpass@http://house/floor4
HTTP-REQUEST
DATA
lamps
lamp lamp
house
floor
@id=4 room
room
lamps
lamp lamp
PERMITTED DATA
http://house/
RESOURCE
/house[./floor/@id=4]///lamps/*
XPath
RESPONSE

Figure 4 (b) Data-Aware Request:

Yes   Apply Changes
Returning 403 (forbidden)
only marked Nodes modified   No
HTTP-REQUEST
POST john.doe:secretpass@http://house/floor4
Body   for $status in /house//lamp/@status return replace value of node $status with "ON"
mod.VDoc   house
floor
room @id=4
room
lamps
lamp lamp
XQuery Update
VDoc   house
floor
room @id=4
room
lamps
lamp lamp
Marking Nodes
VDoc Spec
/house[./floor/@id=4]///lamps
Permission-XPath

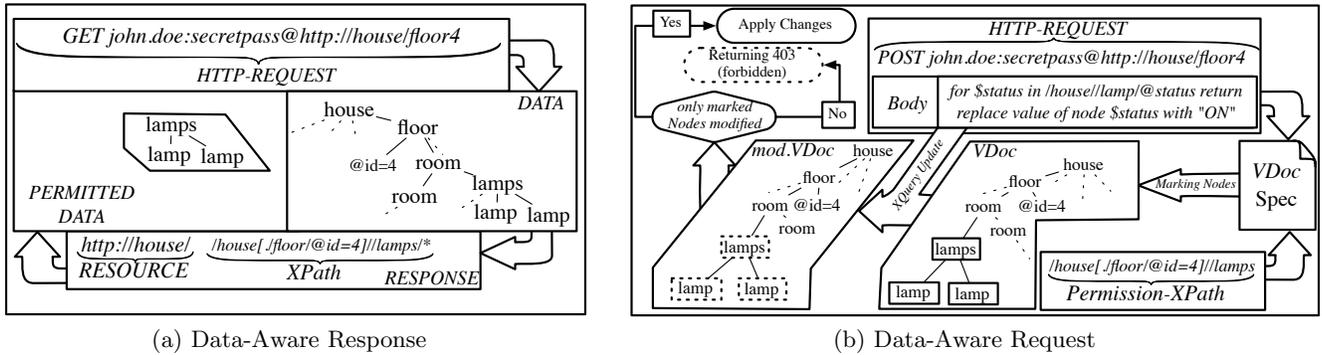(a) Data-Aware Response   (b) Data-Aware Request

Figure 4: Specific Response and Request

is modified within this filter. This finer-granularity handling of permissions is described in detail in Sec. 2.3.

Based on its modular structure consisting of the user model and the *PXD*, HECATE is highly adaptable and flexible. Our focus lies on scalable modifications and fast retrieval of rules which is both ensured due to the tree-structure of the *PXD*. Additional, the independence of the underlying resource is maintained whereas benefits are gathered from optional knowledge about the resources. Within the architecture of HECATE, we satisfy these constraints based on our loose coupling of permissions against resources and users and the optional filtering mechanism with awareness to the layout of the resources. This mechanism is described in detail in the next section.

2.3   Data-Aware Permissions

The indispensable part of the authorization workflow within HECATE relies on resource-independent data only. This includes the URI of the resource, the *HTTP*-verb and the user-credentials as denoted in the section before. Identification of concrete substructures of a resource within this authorization workflow is only supported as long as the substructures are clearly identifiable regarding disjunct URIs and their mapping regarding *data*-nodes to the content. If multiple URIs map to the same resource due to the lack of identification of fixed substructures, the related permissions are able to access all data registered within the related *data*-node. HECATE is without the following extension neither able to distinguish between different substructures of the same resource nor able to offer fine-granular permission sets on these substructures.

The solution to solve this problem is the registration of resource-aware filters. Such filters offer an extension to the common authorization workflow within HECATE and are applied after successful authorization against the requested URI. Since these filters are optional, the common authorization workflow is not constricted even if the appliance of the additional filtering scales with the complexity of the additional filter.

Even if no concrete information about specific datasets must be given, the necessary knowledge for deploying such resource-aware filters includes the kind of the data as well as the layout of the dataset (e.g. the structure of an XML or the kind of columns in CSVs). Based on this knowledge regarding the underlying data, resource-aware filters are established with respect to the representation of the data on the one hand and on a fixed defined purposes of filtering on the other hand.

Based on this layout awareness of the resource, these filters are referenced within *perm*-nodes unbound from any URI. Instead, the referencing takes place over designated identifiers. This enables HECATE to apply the same filters on similar but disjunct resources. Similar to the content referencing within *data*-nodes, *filter*-nodes either contain direct content or a link.

The filtering mechanisms stay as independent as the resource-characteristics, since the filter must adore both, the representation and the independence of different resources. Therefore these filters represent an optional feature which can only be used if knowledge about the underlying resource is present. A concrete workflow of appliances of such methods working with XML-based resources is given within the next Sec. 3.

Since this registration takes place per *perm*-node and therefore per *rule*-node, it is only possible to register exact one filter on each operation. This disables filtering operations on the request and on the response at the same time. Due to the nature of *REST*, read and write accesses are clearly distinguished. This distinction can be mapped on the appliance of the filters.

- *read*-access:
  Filters on *read*-accesses act as direct filters regarding the data-retrieval process resulting not in a modification of the request but an adaption of the response. Consequently, the data is requested based on the URI including possible *REST*-parameters and afterwards filtered with the operation denoted in the corresponding *filter*-node.

- *write*-access:
  Since modification requests return most often simple *HTTP*-codes to denote success or fail, the filtering of the response is not as necessary as an adaption of the content to be written. As a consequence, *write*-accesses are manipulated within the request itself. This results in a possible adaption of the body of a *HTTP*-request regarding the modification of the data.

The different appliance of filters reflected by the different purposes of the related requests fits the authorization workflow of HECATE. Even if this workflow is commonly applicable on all resources, we will show a real-life example based on XML-resources on the next section.

3. XML-BASED RESOURCES

Since XML offers flexible adaption as well as enriched toolsets, multiple non-*REST*-aware resources are encapsulated in XML for convenience reasons[16]. Therefore we chose XML as the base for a real world example including resource-aware filtering.

Listing 1: A resource before modifying

```
1  <house>
    <floor id="1">
3    <room>
      ...
5     <lamps>
       <lamp status="OFF" id="1.1"/>
7      <lamp status="OFF" id="1.2"/>
      </lamps>
9    </room>
    </floor>
11 ...
    <floor id="4">
13   <room>
      ...
15    <lamps>
       <lamp status="OFF" id="4.1"/>
17      <lamp status="OFF" id="4.2"/>
      </lamps>
19    </room>
    </floor>
21 </house>
```

Listing 1 shows an example representing a house. The structure is partitioned into multiple floors including rooms, doors and lamps. Such an example could represent an abstract resource related to the *Web of Things* paradigm where different "things" are simply encapsulated into XML.

Listing 2: XML Fragment denoting read access only

```
1  <resources>
    <rule id="12" perm_id="22"
3    uri="http://house/floor/4"/>
    </rule>
5   <rule id="13" perm_id="23"
     uri="http://house/floor/4"/>
7  ...
    <data uri="http://house/floor/4"">
9    <content>
      /house/floor[@id=4]
11   </content>
    </data>
13 ...
    <filter id="43">
15   <link>
      /house/floor[@id=4]//lamps
17   </link>
    </filter>
19 <perm id="22" filter_id="43"
    verb="get"/>
21 <perm id="23" verb="get"/>
    </resources>
```

Based on this example, List. 2 shows the *PXD* mapping the data of List. 1. As clearly visible, two different rules are mapping the denoted resource. While rule "13" allows the retrieval of all data from the 4th floor, rule "12" filters the same resource by only returning *lamps*-nodes. This additional filtering takes place on the response of the retrieval process and is shown in Fig. 4a. Since the filtering of *lamps*-nodes occurs on the base of an intersected XPath-expression, the resource-aware filtering is scalable as well as extensible and applied after the mandatory checking against URI and

HTTP-operation. Related to modification requests, such a filtering of the data seems not to be as trivial. However, based on the concept of VDocs, we can offer a simple yet effective mechanism applicable to XML.

3.1 The Virtual Documents Concept

Virtual Documents (VDocs) [18] are a general framework for *integrating XQueries into XML documents as computational devices* and processing them efficiently.

As a rough approximation, VDocs are "XML (database) views" analogous to views in relational databases; these are virtual tables in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, VDocs are the results of XQueries computed on demand from the XML files explicitly represented in some storage (like in an XML database or, simply, in a file system), presented to the user as documents. Furthermore they can be presented as file system entities in database or physical files written to a file system. Like views in relational databases, VDocs become very useful abstractions in the interaction with collection of XML documents.

VDocs are defined by a VDoc *Specification* which essentially is a mixture of static XML nodes together with the XQuery queries and rules how the results of those queries should be injected into the result document. VDocs Specs are also parametrizable, that is it may contain certain variables (like URIs of the resources to be processed) that are defined either in VDoc Spec itself or passed on-the-fly upon a VDoc obtaining. Parameters may dramatically change the content of VDoc whereas there only one VDoc Spec exists. As we will see in Section 3.2 the single VDoc Spec may be used to manage modifying REST requests consistently in a fine-granular manner.

Additionally, one of the most advanced features is the ability to *edit* VDocs and *process* the modified version further: changed parts of a VDoc that came from files in our storage will be transparently propagated back to the sources. Editing static parts of a VDoc is not allowed; otherwise a VDoc processor should complain and disallow further processing. Naturally, VDoc XQuery results that are not originated from documents in a storage cannot be edited as well. The strong advantage of editing VDocs is that users can abstract away from the physical documents in the repository and work with semantically consistent objects (like theorems or exercises) focusing only on relevant information aggregated into one logical unit.

3.2 Virtual Documents in Hecate

Given that our resources are in XML, the modifying *HTTP*-verbs (like PUT or POST) naturally may contain XQuery Update statements in the body to modify the requested resource. However, taking into account the presence of fine-grained filtering rules expressed via XPath, certain XQuery Update modifications might be forbidden for a certain user.

To overcome this problem we are proposing to use the VDocs concept together with its editing abilities. Without losing a generality, we will assume that the XQuery Update statements are sent using the POST requests. Also we assume that the POST request are allowed for a certain URI and a user, therefore we should take into account only the request filtering restrictions posed by an XPath expression.

We propose to have a VDoc Spec that admit two parameters: an URI of the resource and the filtering XPath ex-

pression. We call such a VDoc *an authorization VDoc*. Fine-granular editing approach needs several items with respect to VDocs:

1. The VDoc Spec is supposed to fetch the resource identified by the URI and go through all nodes that the XPath expression selects and mark them as *editable*. Not every XPath expression will select at least one node: in this case it will mean that no part of the resource can be modified. Evaluating XPath expressions is not a part of the XQuery specification, however, many XQuery processors provide such a functionality either via extension XQuery functions or a possibility to implement your own external XQuery function in some other programming language. So we consider such a feature as given in our possession.

2. The XQuery Update statement supplied with a POST request will be executed on the VDoc content.

3. A VDoc processor compares the modified and the original VDocs and controls that only parts that were marked editable in step 1 are modified. If a VDoc processor identifies that some nodes that are not *changeable* are modified nonetheless, then it returns an error code to HECATE which in turn sends a forbidden response back to a user. Otherwise, there are two options how to proceed further:

   - Send a modified VDoc with *editing* markers filtered out[2].

   - Send an XQuery Update statement to the underlying system since we can guarantee that this statement would not modify disallowed nodes in the resource.

After HECATE receives a response from the underlying system it can generate the appropriate response for the client.

Despite that the described approach provides fine-granular permissions for modifying resources, it has several disadvantages which might be a good price to pay for the flexibility we gain:

- Typically the processing of VDocs is done in the main memory resulting in scalability problems if the resource is big enough not to fit into the main memory. A possible solution to overcome this problem would be to use an XML database in the HECATE layer.

- Fine-granular editing of the resource implies two processing steps: (i) a modification of the resource in HECATE, and (ii) its modification in the underlaying system. It might be not so efficient as doing the modification in only one software component. On the other hand, it allows us to maintain the loose coupling between our authorization framework and the underlying system.

---

[2]Those markers could be some auxiliary attributes embedded into the XML elements, special comment nodes or some kind of processing instructions – this depends on a VDoc processor implementation

3.3 A Modifying Example

Let us consider a simple example of a data-aware request where the related workflow is shown in Fig. 4b. Assume that a user is allowed to modify the state of the lamps only on the fourth floor of a particular house resource (the filtering XPath would look like /house/floor[@id=4]//lamps), and all lamps in the house are initially off as denoted in List. 1.

Now assume that the user sends a request with an XQuery Update statement that intends to switch all lamps on:

```
for $status in /house//lamp/@status return
  replace value of node $status with "ON"
```

The authorization framework first supplies the URI of the requested resource together with a filtering XPath expression to an authorization VDoc Spec (see Section 3.2). A result VDoc will have elements amenable to modifications marked with special VDoc attributes [3]. After the content of VDoc is retrieved, a supplied XQuery Update expression is applied to it (see List. 3).

Listing 3: A *marked* and modified resource

```
<house>
2  <floor id="1">
     <room>
4    ...
     <lamps>
6      <lamp status="ON" id="1323412"/>
       <lamp status="ON" id="5456"/>
8    </lamps>
   </room>
10  </floor>
   ...
12  <floor id="4">
     <room>
14   ...
     <lamps>
16     <lamp vdoc:uri="..."
         vdoc:xpath="/house[1]/floor[4]/room[1]/lamp[1]"
18       status="ON" id="3443"/>
       <lamp vdoc:uri="..."
20       vdoc:xpath="/house[1]/floor[4]/room[1]/lamp[2]"
         status="ON" id="5456"/>
22    </lamps>
   </room>
24  </floor>
</house>
```

Note that the status of *not marked* lamps (for the floor 1) have also been modified. A VDoc processor will compare it with an original VDoc (where all lamps are off) and will recognize that the statuses of not marked lamps have also been changed, thus it means that the modification was not allowed for every XML node that has been altered. Therefore a forbidden request is sent back to the user. If there were only lamps on the fourth floor that we modified then a VDoc processor would successfully validate changes and would send a POST request with a modified resource to the underlying system filtering out the marker attributes beforehand.

4. IMPLEMENTATION

HECATE consists out of modular components enabling integration into already existing projects. We therefore prove

---

[3]In current VDoc implementation those attributes denote the URI of a document and the XPath location of an element inside the document. Strictly speaking, such detailed information is not necessary for our scenario, however, this marking stays consistent with a general VDoc editing approach.

the practicability of HECATE within two independent projects namely JAX-RX [11] and TNTBASE [17]. While JAX-RX represents a common layer for equipping XML databases with uniform *REST*-functionality, TNTBASE represents a native XML database system itself. Even if the purpose of both projects is different, HECATE equips both systems with authorization. Within both implementations we prove that the idea of HECATE is easy implementable within current infrastructures. Furthermore, with JAX-RX, we extended our common layer for XML resources with the functionality so third-party XML based projects can make direct use of our approach. JAX-RX as well as TNTBASE are both available as free open-source projects.

5. CONCLUSION

HECATE enables *REST*-resources to be guarded within an integrative authorization management. The access control stays independent from the resource representation and is based on *HTTP*-operations. With optional knowledge about the underlying resource, HECATE is able to offer filters and operations with finer granularity for the resources. We proved our approach with an implementation included in JAX-RX and TNTBASE and showed the practicality of our independent authorization model

HECATE enables a central permission management for multiple resources. The resources must not be aware of each other neither must HECATE be aware of the characteristics of the resources. Even if the *HTTP*-operations represent the base for our permission model, HECATE supports additional resource-aware filtering. This filtering must occur under awareness of the resource from which the response is filtered.

Open problems include the mapping of resource-bound user credentials into the HECATE authorization framework. Another mapping regarding HECATE users and resource-user is necessary to solve this issue.

The next steps include the awareness of recursive resources. Based on the tree-structure of the *PXD*, an order-awareness of nested resources increase the granularity of the authorization. Furthermore we want to exploit the usage of finer filters. Since the current permission model for requests relies only on *HTTP*-verbs, we believe that the already proposed filtering of the responses and the requests offers multiple areas of future work especially regarding the representation of collections and temporal views of resources.

6. ACKNOWLEDGEMENT

7. REFERENCES

[1] R. Bhatti, E. Bertino, and A. Ghafoor. A trust-based context-aware access control model for web-services. *Distributed and Parallel Databases*, 18(1):83–105, 2005.

[2] R. Bhatti, J. B. Joshi, E. Bertino, and A. Ghafoor. Access control in dynamic xml-based web-services with x-rbac. Technical report, Purdue University, 2003.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and T. B. Textuality. Extensible Markup Language (XML) - version 1.0, 1997.

[4] J. Clark and S. DeRose. XML path language (XPath) version 1.0, 1999.

[5] D. Crockford. The application/json media type for javascript object notation (json), 2006.

[6] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. In *ACM Transactions on Information and System Security (TISSEC)*, 2002.

[7] E. Damiani, S. D. C. di Vimercati, and P. Samarati. Towards securing XML Web services. In *ACM workshop on XML security*, 2002.

[8] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *ACM SIGMOD international conference on Management of data*, 2004.

[9] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[10] A. Gabillon. An authorization model for XML databases. In *ACM Workshop on Secure Web Services*, 2004.

[11] S. Graf, L. Lewandowski, and C. Grün. JAX-RX, unified REST access to XML resources. Technical report, University of Konstanz, 2010.

[12] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. *Architecting the Internet of Things*, chapter From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices. Springer, 2010.

[13] S. Hada and M. Kudo. XML access control language: Provisional authorization for XML documents. Technical report, IBM Research Research Laboratory, 2000.

[14] D. Peng, C. Li, and H. Huo. An extended UsernameToken-based approach for REST-style Web Service Security Authentication. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, 2009.

[15] H. Story, B. Harbulot, I. Jacobi, and M. Jones. FOAF+ SSL: RESTful authentication for the social web. In *First Workshop on Trust and Privacy on the Social and Semantic Web (SPOT2009)*, 2009.

[16] E. Wilde. Putting things to REST. Technical report, 2007.

[17] V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Balisage: The Markup Conference 2009*, 2009.

[18] V. Zholudev and M. Kohlhase. Scripting Documents with XQuery: Virtual Documents in TNTBase. In *Balisage: The Markup Conference 2010*, 2010.