Paper for the Seventh International Workshop on Web Information Systems Modeling (WISM 2010),
November 1-4, 2010, Vancouver, BC, Canada

Integrity Assurance for RESTful XML

Sebastian Graf[1], Lukas Lewandowski[1], and Marcel Waldvogel[1]

Department of Computer and Information Science, University of Konstanz
78457 Konstanz, Germany
{Sebastian.Graf,Lukas.Lewandowski,Marcel.Waldvogel}@uni-konstanz.de

Abstract. The REpresentational State Transfer (REST) represents an extensible, easy and elegant architecture for accessing web-based resources. REST alone and in combination with XML is fast gaining momentum in a diverse set of web applications. REST is stateless, as is HTTP on which it is built. For many applications, this not enough, especially in the context of concurrent access and the increasing need for auditing and accountability. We present a lightweight mechanism which allows the application to control the integrity of the underlying resources in a simple, yet flexible manner. Based on an opportunistic locking approach, we show in this paper that XML does not only act as an extensible and direct accessible backend that ensures easy modifications due to the allocation of nodes, but also gives scalable possibilities to perform on-the-fly integrity verification based on the tree structure.

1 Introduction

1.1 The Multiple Facets of XML

The eXtensible Markup Language (XML) [2] represents one major paradigm in nowadays *WWW* environments. Not only used as a quasi standard when it comes to configuration issues and handling of meta information, XML is also used as a direct data source regarding the preparation and visualization of information. Famous representatives for these use cases are XHTML as well as SVG or KML. These different XML dialects show the necessity of human-readable file formats, and, accompanied with an enriched tool-set like XPath [8], XQuery/Update [5], and XSLT [9], highlight the applicability in many different areas.

Another perspective to the evolution of XML as a direct accessible data storage format can be observed in modern storage systems. Not only have modern common (object-)relational database systems the ability to store and retrieve native XML. The ease of use, flexibility, and adaptability gave also birth to several non-relational databases [15] that indeed have an essential reason to exist nowadays.

Besides the utilization of XML as a data-format backend in visualizations and storage applications, XML is also used for providing integrated unified access regarding entire workflows of querying and modifying data, especially in the *WWW*. Apache Cocoon [23] and XForms [3] are main represents, along multiple others, when it comes to an all-in-one XML based solution of retrieving, transforming and presenting data.

1.2 Stateless Access to Resources with REST

The *REpresentational State Transfer* [11] constitutes a new and elegant approach to access distributed resources. Instead of encapsulating requests in containers like SOAP, REST is accessing resources directly in a stateless manner: No session handling and transaction check is performed, each request is encapsulated, atomic and bound to a direct resource.

This easy way of handling and accessing distributed resource as well as the clean definition of methods to interact with these distributed resources are the ingredients for the success of REST.

The usage of REST is defined on three independent axis, (a) the REST *verbs*, (b) the REST *resources*, and (c) the REST *parameters* (cf. (1)). Every request is bound to a *verb*. The *verb* determines the kind of action related to the requested resource and the corresponding *parameters*. REST *verbs* are `POST` requests to create/append a resource, `PUT` requests to place new content on a given resource (e.g. update), `DELETE` requests for removal operations and `GET` requests for common read-only access. These simple operations defined in HTTP are the key ingredients of a RESTful application.

Besides the *verbs*, REST is based on direct accessible *resources* which are direct parts of the URI. A *resource* is always a concrete manifestation of data which can be accessed with all REST verbs in a similar way.

The third important part of REST are the *parameters*. *Parameters* can contain any meta information for accessing data, for instance queries or additional commands. They are always optional and bound to a *resource*. Thus, a *parameter* can be used to filter or adapt the operation performed on a REST *verb*.

$$\underbrace{GET}_{Verb}\underbrace{http://host/data.xml}_{Resource}\underbrace{?query = descendant - or - self :: x}_{Parameter} \tag{1}$$

The URL above shows a simple REST access on XML. Obviously, REST matches perfectly with XML as resource handler. Due to the architecture of the requests, any tree-like structure can be accessed directly without any limitations via any URI.

XML equipped with unique identifiers, either based on node-levels like ORDPATH [18] or on any tree-based encoding, is able to answer requests directly on the node-level as well as on the XML itself. This enables REST to access XML substructures directly via *resources* without any implementation of necessary *parameters* since direct node access has to be based otherwise on queries for example.

1.3 Contribution and Problem Statement

RESTful access to resources is easy to provide and offers a high flexibility in its utilization. However, the simplicity of REST comes at a price: Considering multiple, concurrent modifications on the resource, a RESTful backend can render the semantic state of the resource invalid if an operation can not be performed in an atomic manner and therefore must be split into multiple consecutive requests. Besides, as REST is a stateless technique, any session or transaction based approach will not satisfy the paradigms of a RESTful application. To the best of our knowledge, there is currently no approach that is able to check the entire integrity of a RESTful resource against consecutive requests from disjoint clients. Nevertheless, we believe that such an integrity check increase the usability of REST without degenerating it to a non-stateless approach.

In this paper we propose a technique based on opportunistic locking to provide data integrity on tree-structured data regarding the handling of consecutive as well as concurrent REST requests. First, we generate a checksum for the tree based on *Merkle Trees* [16]. These checksums are the key for integrity checks for any RESTful access on XML resource. Thus, we enable RESTful applications to verify the integrity within each request while adhering to the stateless paradigm of REST.

1.4 Related Work

The set of REST and XML has been explored in various ways. Wilde [21] encourages the usage of REST even for not web-related resources. Objects that cannot be represented as web content are encapsulated in XML to provide a common way to be accessed. [19] describes an approach about the XML dialect *BPEL* that is capable of acting RESTful. This approach fits perfectly in our use case of transaction integrity checking. Kramis et al. [12] described an approach which allows the access of any XML document in a common way. However, even if temporal aspects are considered no integrity check is performed. The common access of XML data is also described in [1]. For communicating between fixed and mobile clients, XML-RPC is used for a session based approach. [20] enables REST to work transactional based on allocations of transactions as separate resources. This approach works directly on single resources only but could result in race conditions.

As our approach utilizes the structure of an XML resource, that corresponds to disjoint subtrees as well as the direct allocation of nodes, we evaluate possibilities to ensure structural integrity of tree structures. Based on the *Merkle Trees* [16], there are multiple different approaches to ensure integrity [4]. All of these approaches make use of recursive structural computations. Checksum methods employing the same idea can be used to provide integrity in our XML structures.

Validation approaches that are directly related to XML are not solely structure based. Some of them utilize a scheme based validation [10]. However, since we focus on concurrent operations on the nodes, a check against a DTD does not satisfy our needs. Even concurrent updates can result in a valid XML which is not valid in the semantic context of the sequential requests.

[7] uses XML as a base for defining a language that provides data integrity. However, stateless communication is not considered an alternative in this approach. [6] improved this approach to check the integrity of distributed web communication systems. This system neither relies on stateless communication,

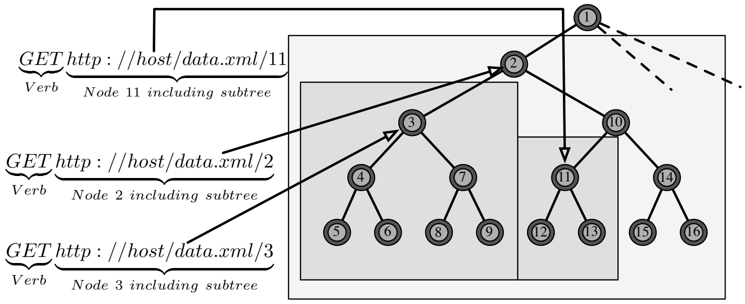


Fig. 1: RESTful access to tree

nor does it utilize any additional information from the underlying resource that in turn could contain benefiting information. Finally, [22] proposes a protocol based approach to provide security and integrity of the retrieved data.

2 Integrity Check for REST-enabled XML Resources

The verification of integrity regarding consecutive REST accesses on tree based data is mainly based on two aspects: First, the definition of a RESTful access to the resource in a way that it can explore the tree structure in a native manner and second, an integrity check of the tree structure including a checksum based resource allocation scheme.

2.1 Unified RESTful access to tree based structures

The motivation for our verification approach is to ensure concurrent data accesses while adhering to the strict stateless architecture defined by REST. Besides, we additionally integrate full RESTful paradigms in our approach. These paradigms are represented within our URI specification as follows:

- A URI can request one XML by its resource name. In that case the root node including the entire tree is returned.
- Each node in the XML can be accessed with a unique identifier (similar to Temporal REST [12] or ORDPATH [18]). If a resource is offering such a feature, the unique identifier can be accessed over REST as a direct resource as well. The choice of the encoding of the unique identifier is independent from our approach. In case of node-level access, the desired node plus the underlying subtree is returned.

Figure 1 shows document accesses based on our definition. Obviously, coupling REST requests with unique identifiers per node is straightforward. However, it is important to understand that requests are only valid for the requested node as well as the related substructure. As our approach utilizes the structure of the tree to perform integrity verification, requests coupled to one node are not allowed to access ancestor nodes or the corresponding subtrees. Therefore, related to REST parameters, that for instance can contain XQuery/Update [5], we only allow the usage of the forward axis in the related subtree. However, this is not a constraint at all as most of the modification and query languages rely on XPath, and each XPath expression can be evaluated only by utilizing the forward axis [17].

2.2 Integrity check of the tree

As we are working with XML, we utilize the tree structure not only in terms of the direct allocation of nodes as resources. When it comes to on-the-fly verification of the integrity of the XML resource we rely on recursive algorithms [16] to generate checksums for each node. (2) below denotes the structure of the checksummed tree.

$$n.hash = \mathrm{H}(\mathrm{H}(n.content) \parallel n.child(0).hash \parallel n.child(1).hash \parallel \ldots) \qquad (2)$$

Thereby, n represents the node and $H(x)$ is a hash function with input x.[1]

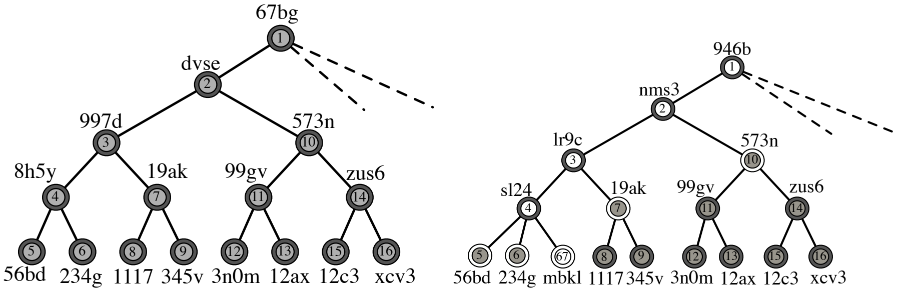
The checksum of a node relies on its hash value and therefore is defined as the hash value of the content of a node combined with the hash value of all of its child nodes. The selection of the specific hash algorithm itself can be adapted to the specific use case of the resource in the application: If the resource has to be responsive, a fast hash function should be chosen. If the structure is in need of high integrity, a more stable hash function should be considered. The approach itself is as stable as the used hash function.

Figure 2a shows such a checksummed tree structure. Checksums are generated based on a recursive relation where each node inherits the integrity of its corresponding subtree. Therefore, the complete subtree rooted at a node can be verified in a single step by checking the node's hash value. Projected on the already described RESTful access where one resource can be a XML tree as well as a qualified node, the checksum of a node is guarding the entire integrity of a resource node plus the underlaying subtree.

Any modification related to the structure of a (sub-)tree or the content of a single node results in the regeneration of the corresponding checksums. Fig. 2b shows an example for the checksum regeneration while algorithm 1 describes the algorithm. Each time a request is performed, the checksum delivered with the request is compared to the one of the requested resource. If both checksums differ, an error is returned. In case of a modification in the tree, all checksums on the path to the root are recomputed with the help of the corresponding siblings.

In the example of Fig. 2b, the white node labeled 67 is the node which is inserted in the tree. The nodes labeled 5, 6, 7 and 10 (depicted in grey with a white border) are only touched for read operations. These read operations are necessary to perform the update of the parent nodes 4, 3, 2 and 1 since the checksums are always based on the checksums of the related children as well.

[1] To improve processing overhead for nodes with high degree, the sequence of children can also be internally structured and hashed into a hierarchy.



(a) Checksummed Tree

(b) Modified Checksummed Tree

Fig. 2: Recursive regeneration of checksum while inserting new node as a leaf

This example highlights that, due to the recursive structure, only the checksums of the nodes on the path to the root need to be modified during an update operation. In the worst case all nodes on the path to the root need to be updated depending on the modified leaf. Obviously, the cost of a modifying access to the tree corresponds to the height of the tree. Therefore, as we only need to update the checksums on the path up to the root, we are able to do an on-the-fly update of the checksums while traversing the tree from the node that was modified up to the root of the tree.

It is important to understand that we make use of the structure of the data. Since all requests occur on the tree, we do not have to regenerate every checksum of the entire data space within single modifications. All nodes that are not directly affected by a modification request are excluded from any updating mechanisms as long as they are located in disjoint subtrees.

2.3 Request-based integrity validation

The structure of the tree itself carries the prove of integrity at every point in time. Therefore and due to the atomicity of REST requests we can provide validation of consecutive REST requests that access the same resource. After each request the checksum of the requested resources is returned. Thus, if the request is based on an entire XML tree, the checksum of the root node and the current status of the entire tree is returned to the client. If a request affects a node resource, the checksum of the subtree rooted at the corresponding node is returned to the client. As communication base for the checksum within the REST requests, the *ETag* field of the HTTP specification is used.

Validating the expected state of a resource is often related to previous, consecutive requests/modifications on the same resource: A client requests resources, checks the delivered data and tries to perform subsequent operations on it. With the first request, a checksum of the requested resource – that can be the entire XML as well as a substructure based on an unique node – is delivered together with the requested data in the *ETag* field of the HTTP header. This checksum is returned to the server in the consecutive request. If the request is modifying the data, a new checksum is computed and again the checksum is returned with

Algorithm 1: Handle Request

```
Input: HTTPRequest $request$, Hash function $H$
Output: HTTPResponse $response$
begin
    Node $n \longleftarrow request.resource$
    if $request.checksum = n.checksum$ then
        $opReturn = opOnData(n, request.verb, request.parameter)$
        if $opReturn \neq wasValidOp$ then
            $response \longleftarrow$ new ErrorResponse($opReturn.errorCode$)
        else
            if $opReturn = wasModifyingOp$ then
                Node $m \longleftarrow n$
                repeat
                    $m.checksum \longleftarrow H(content(m)) \parallel h$
                    $h \longleftarrow m.checksum$
                    for $r \in m.siblings$ do
                        $h \longleftarrow h \parallel H(r)$
                    $m \longleftarrow m.parent$
                until $m \neq root$
                $m.checksum \longleftarrow H(content(m)) \parallel h$
            $response \longleftarrow$ new SuccessResponse(200, success, n.checksum)
    else
        $response \longleftarrow$ new ErrorResponse(412, Precondition failed)
    return $response$
end
```

the following response. If the server observes a different checksum for the same requested resource, the HTTP error *412 (Precondition Failed)* is returned to the client thus the client is informed about the concurrent modification of the data.

Figure 3 depicts a consecutive, concurrent check-then-act situation that highlights our approach. First, client 1 gets the resource with id 3 in the tree. The hash value *997d* is returned to the client together with the requested resource. A second request is performed by client 2 but on the node with id 4. This node is a child of the node requested by client 1. The returned checksum for this node is *8h5y*.

Subsequently, client 1 performs a POST operation to insert a new leaf in the subtree maintained by client 1. The new node has the id 67 and all the checksums on the path to the root are updated before the request is solved and a suitable HTTP code which depict success of the operation is returned to the client. The related new checksum (*lr9c*) of the requested resource with id 3 is returned to the client in the *ETag* of the response. The corresponding HTTP communication protocol is listed in table 1a.

In the meantime, client 2 tries to access the resource with id 4. Since the request is shipped with the checksum of its last request *8h5y*, the server compares the two checksums from the concurrent requests on the same resource (*8h5y* and

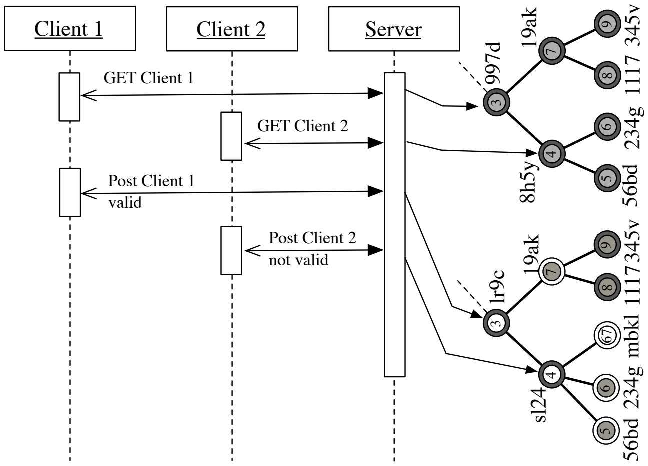


Fig. 3: Concurrent REST requests

sl24). Due to the modification of the same subtree within the request of client 1, the checksum of node 4 changed. Therefore, the request from client 2 is denied due to the disparity of the checksum shipped with the request and the checksum currently associated with node 4. Client 2 first has to become aware of the changes in the data and retrieve the new checksums for the requested resources before new checksum-guarded requests become valid. The HTTP communication protocol corresponding to this is listed in table 1b

This example workflow shows that our approach ensures data verification in a RESTful manner. As long as clients only request the subtree of a specific node, our integrity approach can even handle multiple accesses. Furthermore, our checksum approach can easily be modified in a kind that within every request/response a checksum is shipped without direct interaction from the server site. This would enable clients to perform checking against data integrity by themselves. If derived checksums differ, the client has to find out what concurrent operation was modifying the related resource or the underlying subtree.

3 Conclusion and Future Work

The proposed setup of REST and XML was implemented using *JAX-RX* [14] as interface and *Treetank* [13] as well as *BaseX* [15] as XML resource. Our implementation substantiates our assumption that our approach leverages trust in stateless data handling by a simple though powerful validation mechanism by resolving the lack of confidence in the data accessed over REST with an in-data integrity verification.

Regarding further extensions of our approach, we believe that our approach can make use of more sophisticated hashing adaptions as well. With an intelligent hashing strategy it is possible to reduce the overhead of adapting the hash values in a tree every time a modification occurs even though these modifications

Table 1: Example of concurrent HTTP communication

(a) Communication for client 1

HTTP Request	HTTP Response
GET http://. . . /3	
	ETag(997d) <node> . . . </node>
POST *ETag(997d)* http://. . . /3/firstChild <node> . . . </node>	
	ETag(lr9c) 201 CREATED

(b) Communication for client 2

HTTP Request	HTTP Response
GET http://. . . /4	
	ETag(8h5y) <node> . . . </node>
POST *ETag(8h5y)* http://. . . /4 <node> . . . </node>	
	ETag(sl24) 412 PRECONDITION FAILED

are rather small. Furthermore, since we are restricting our access at the moment to fixed resources and therefore substructures in tree, we want to increase the flexibility of our approach regarding the computation of checksums and concurrent accesses in the tree to track consecutive requests on different nodes. To provide such auditing features, we plan to equip our approach with a versioned backend to track consecutive modifications on distributed data. This envolves even more power to our integrity approach since with every modification the related integrity structure can be secured as well.

Checking the integrity of accessed data is one of the most important tasks regarding distributed applications. Although REST offers a great flexibility we believe that we can ensure the confident access to the data in a way that is not restricting REST but gives the possibility to overcome the uncertainness of stateless data access.

References

1. Alvarez-Cavazos, F., Garcia-Sanchez, R., Garza-Salazar, D., Lavariega, J.C., Gomez, L.G., Sordia, M.: Universal access architecture for digital libraries. In: Conference of the Centre for Advanced Studies on Collaborative Research (2005)
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Textuality, T.B.: Extensible markup language (xml) - version 1.0 (1997)
3. Cardone, R., Soroker, D., Tiwari, A.: Using xforms to simplify web programming. In: International Conference on World Wide Web (2005)
4. Carminati, B., Ferrari, E., Bertino, E.: Securing xml data in third-party distribution systems. In: ACM International Conference on Information and Knowledge Management (2005)
5. Chamberlin, D., Florescu, D., Robie, J., et al.: XQuery update facility (2006)
6. Chi, C.H., Liu, L., Yu, X.: Data Integrity Related Markup Language and HTTP Protocol Support for Web Intermediaries. Embedded and Ubiquitous Computing (2006)

7. Chi, C.h., Wu, Y.: An xml-based data integrity service model for web intermediaries. In: International Workshop on Web Content Caching and Distribution (2002)
8. Clark, J., DeRose, S., et al.: XML path language (XPath) version 1.0 (1999)
9. Clark, J., et al.: XSL transformations (XSLT) version 1.0 (1999)
10. Fan, W., Libkin, L.: On xml integrity constraints in the presence of dtds. Journal of the ACM (2002)
11. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
12. Giannakaras, G., Kramis, M.: Temporal REST—How to really exploit XML. In: IADIS International Conference WWW/Internet (2008)
13. Graf, S.: Treetank, a native xml storage. Tech. rep., University of Konstanz (2009)
14. Graf, S., Lewandowski, L., Gruen, C.: Jax-rx, unified rest access to xml resources. Tech. rep., University of Konstanz (2010)
15. Holupirek, A., Grün, C., Scholl, M.H.: Basex and deepfs joint storage for filesystem and database. In: International Conference on Extending Database Technology (2009)
16. Merkle, R.C.: A digital signature based on a conventional encryption function. In: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (1987)
17. Olteanu, D., Meuss, H., Furche, T., Bry, F.: Symmetry in xpath. In: EDBT Workshop on XML Data Management (2002)
18. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: Ordpaths: insert-friendly xml node labels. In: ACM SIGMOD International Conference on Management of Data (2004)
19. Pautasso, C.: Bpel for rest. In: International Conference on Business Process Management (2008)
20. da Silva Maciel, L.A.H., Hirata, C.M.: An optimistic technique for transactions control using rest architectural style. In: ACM Symposium on Applied Computing (2009)
21. Wilde, E.: Putting things to REST. Tech. rep. (2007)
22. Yao, D., Koglin, Y., Bertino, E., Tamassia, R.: Decentralized authorization and data security in web content delivery. In: ACM Symposium on Applied Computing (2007)
23. Ziegeler, C.: Cocoon: Building XML Applications. Pearson Education (2002)