R2SMA - A Middleware Architecture to Access Legacy Enterprise Web Services using Lightweight REST APIs

Jan Königsberger and Bernhard Mitschang
Institute for Parallel and Distributed Systems, University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany

Keywords: REST, API, SOAP, Web Service, SOA, Enterprise SOA, Architecture.

Abstract: This paper presents the SOAP-to-REST Middleware Architecture which provides an abstraction layer for conventional web services. This layer semi-automatically creates REST API (REpresentational State Transfer) proxies for existing enterprise web services that allows companies to provide flexible and lightweight access to exiting web services. Therefore, REST APIs can be offered for existing web services without the need to adapt them, which allows for flexible and fast integration scenarios. The architecture also provides additional enterprise-grade functionality such as caching and security.

1 INTRODUCTION

Service-oriented architectures have been the method of choice for enterprise application integration in recent years. Lately, the focus has shifted from large enterprise applications providing ”conventional”, usually SOAP-based, web services to a more diverse and heterogeneous integration landscape. This includes mobile applications, microservices and devices as part of the (Industrial) Internet of Things (IIoT).

While conventional web services provide many benefits and possibilities required for adoption within an enterprise environment, like reusability, ease of integration or fast adaptability, companies are now faced with new technological challenges. New and more lightweight use cases from the mobile and IIoT context also require more lightweight integration technologies like REST (REpresentational State Transfer, (Fielding, 2000)), which allows for a fast and flexible access to resources and information. At the same time new requirements emerge that ask for the use of existing applications within those new use cases. With the ongoing digitalization, companies also start to expand into new business areas, for example by providing access to their services to outside developers via pay-per-use scenarios. Therefore, companies need to find a way to make a large, already existing service portfolio available via more lightweight protocols. The development of new services and APIs is, of course, a huge effort. Aside from the development of new interfaces, the applications also have to be tested and redeployed, which may lead to undesired and costly downtimes.

To help alleviate these challenges, this paper presents the SOAP-to-REST Middleware Architecture. The architecture abstracts from the existing legacy service layer and provides a semi-automatic way to create REST proxies for existing conventional web services, without the need to adapt the existing services. Current approaches in this area are mostly manual processes and lack certain enterprise-grade functionality such as security and caching.

2 REQUIREMENTS

This paper aims to design an architecture that supports companies in providing RESTful access to their legacy SOA infrastructure. From an enterprise viewpoint certain requirements have to be fulfilled by the architecture to be applicable in practice, especially regarding security concerns. This section discusses those requirements in detail.

R_1**: Reuse of Existing Services.** Having relied on service-oriented Architectures (SOA) for some time, companies usually have a large number of conventional web services available and integrated into their everyday processes. To have all of these services ported to RESTful APIs by application developers would require enormous efforts in time and money, both of which are notoriously scarce resources within an enterprise. This makes it necessary to find a way to

Königsberger, J. and Mitschang, B.
R2SMA - A Middleware Architecture to Access Legacy Enterprise Web Services using Lightweight REST APIs.
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018) - Volume 2*, pages 704-711
ISBN: 978-989-758-298-1

provide RESTful access to those services without the need for each application to develop new REST APIs and without the additional effort that comes with testing and redeployment of all applications.

R_2**: Flexible and Quick Access.** When providing mobile access to services, a fast time to market is important. Development cycles in the "mobile world" run a lot faster than in the slower changing enterprise world. Therefore users expect regular updates and quick improvements. Especially large companies often struggle initially to meet these expectations.

Historically, companies have developed applications in relatively long release-cycles, for example only having one or two releases per application and year. More agile development models such as Scrum and DevOps are only slowly finding their way into enterprise software development. Therefore, an architecture is necessary that allows to handle both the fast paced changes in RESTful APIs as well as those of the slower back-end applications. McKinsey coined the term *two-speed architecture* (Bossert et al., 2014) to describe such architectures. This need for flexibility and fast access to existing services via lightweight interfaces therefore requires the decoupling of APIs from the back-end applications.

R_3**: Reliability.** Reliability is always one of the most important goals within enterprise architectures. With our goal of providing RESTful access to existing SOAP-based web services, reliability can be viewed in three different dimensions:

(1) REST doesn't provide reliable messaging as we know it from conventional web services, which provide reliable delivery on the transport level, e.g. through WS-ReliableMessaging[1]. Indeed, this is not necessarily required and also could be handled within the business logic.

(2) Access to the services has to be reliably available, i.e. services should never be offline or not accessible, for example by setting up fail-over systems.

(3) Consumers accessing RESTful services expect the interfaces to adhere to the REST constraints. This includes the correct behavior of a system when querying it using specific HTTP methods. This style also differs from what developers know from conventional web services. The main difference is that conventional web services are operation-based, whereas REST APIs are resource-centric.

[1]http://docs.oasis-open.org/wsrx/wsrm/200702/wsrm-1.2-spec-os.html

R_4**: Security.** Within an enterprise context there usually are restrictions and security mechanisms in place to prevent unauthorized users to access data and information. In case of web services, this usually means authentication mechanisms such as user name/password or API keys. These security requirements also have to be fulfilled by a REST proxy. Therefore, the proxies have to be tied into the overall enterprise security architecture of a company and appropriate authentication information has to be propagated to the back-end services.

R_5**: Versioning.** When creating and providing web services to users, these services usually will change over time. If the functional range of an application is extended or modified, these changes usually also need to be reflected in the service interfaces. Adapting consumer applications to a new interface typically takes some time and it may not be possible to immediately switch to a new interface version for all consumers. This makes it necessary that different versions of an interface might exist in parallel. It also might be necessary to only change the RESTful interface of a service for some reason, for example to implement new architectural guidelines. Such changes should ideally be handled without any modification of the underlying back-end application.

3 RELATED WORK

We did a comprehensive analysis of current approaches for RESTful access to conventional web services. Work in this area can be clustered into two major areas:

- *Scientific research papers*, which mostly illuminate the differences between conventional web services and REST APIs (e.g. (Pautasso et al., 2008)). There is very little work researching RESTful access to conventional web services.
- *API middleware tools and corresponding industry whitepapers*: There are some products, mainly API middleware solutions, which already support the creation of REST to SOAP proxies to some extent.

The relevant scientific papers are mainly motivated from a workflow-oriented viewpoint. In (de Giorgio et al., 2010) the authors look into how both SOAP-based and RESTful web services can be used within BPEL workflows. They describe a way to semantically map the description of both services types, allowing them to switch between a SOAP-based and a RESTful web service at runtime.

The reverse direction to our approach is presented in (Peng et al., 2009). Their SOAP2REST framework allows the wrapping of RESTful web services as SOAP-based services to utilize them within BPEL processes. The Framework relies on a WADL description (Web Application Description Language) of the RESTful service to generate a SOAP proxy.

Apigee (Apigee, 2014) provides the users of its API platform with a wizard that helps them to convert SOAP-based into RESTful web services. To achieve this, the wizard utilizes the WSDL description (Web Service Description Language[2]) of the service to generate a skeleton for the RESTful service and lets users define the required REST parameters like resource URIs and HTTP method. In comparison to our approach, functionality for enterprise-readiness such as caching or security-propagation don't seem to be available.

Mulesoft (Agrawal, 2015) describes a mostly manual process to create a REST proxy. After defining the desired REST description using RAML, the vendor's *Anypoint Studio* is used to create a message transformation for a HTTP query to a SOAP request.

To summarize, none of the existing solutions fully address the requirements we presented in Section 2. We therefore present our REST-to-SOAP Middleware Architecture in the following sections.

4 REST-to-SOAP MIDDLEWARE ARCHITECTURE

Conventional web services have many benefits that make them the ideal technology to integrate enterprise applications. With the growing demand for greater flexibility within enterprise architectures in recent years, more lightweight technologies, such as REST, have been integrated into these architectures.

The REST-to-SOAP Middleware Architecture (R2SMA) we present in this section allows a company to abstract existing conventional web services via a middleware proxy that provides REST APIs for those back-end services. Figure 1 shows the R2SMA. The core functionality is handled by the REST-to-SOAP-Proxy component, which provides REST endpoints for the conventional web services to consumers. The component also provides caching as well as transformation capability for REST requests. Existing web services are provided via a middleware layer, like an enterprise service bus, that is connected to the transformation sub-component. The architecture is completed by an enterprise security gateway dealing with authentication and authorization as well as a repository component for interface management and proxy generation.

[2]https://www.w3.org/TR/wsdl20/

4.1 Transformation

Each request to a REST API proxy triggers a subsequent request to a back-end web service, unless the response is already stored in the cache (see next section). This second request is completely transparent to a service requester. The transformation component translates a message from a REST-compatible JSON representation into a data format understood by the back-end service, usually SOAP. The specific transformations and data models are defined during the creation of a REST proxy service and are part of its description document. The transformation component also validates the data in the incoming requests to make sure they can be mapped to the SOAP format. A thorough validation is necessary, because JSON only supports limited data types. For example: numbers are not differentiated between integer and floating-point values, which may lead to undesired behavior.

4.2 Caching

Caching is required to manage the load on back-end systems and to provide faster response times. RESTful web services are in certain common usage scenarios required to process more requests than conventional ones to achieve the same result. This is due to the fact that REST operates on resources. Retrieving a list of items and all their data following the REST paradigm consists of 1) retrieve list of items and 2) query each item (i.e. resource) separately for its information. This means the longer the list, the more requests are necessary. Conventional services usually return all items within a single response. Additionally, REST supports *content ranges*, i.e. the requester can define to only return a certain range of items from a list. Without any caching mechanism this means that if a requester splits the retrieval of a list containing 500 items into chunks of 50 items, 10 requests are necessary to retrieve all items. In other words: Each of the 10 requests is propagated to the underlying back-end service, increasing the load.

The following sections discuss ways to design the caching component for our architecture.

4.2.1 Cache Security

In any enterprise environment users oftentimes work with highly sensitive data. This on the one hand necessitates that a cache is included in the overall security concept, ensuring that users have only access to

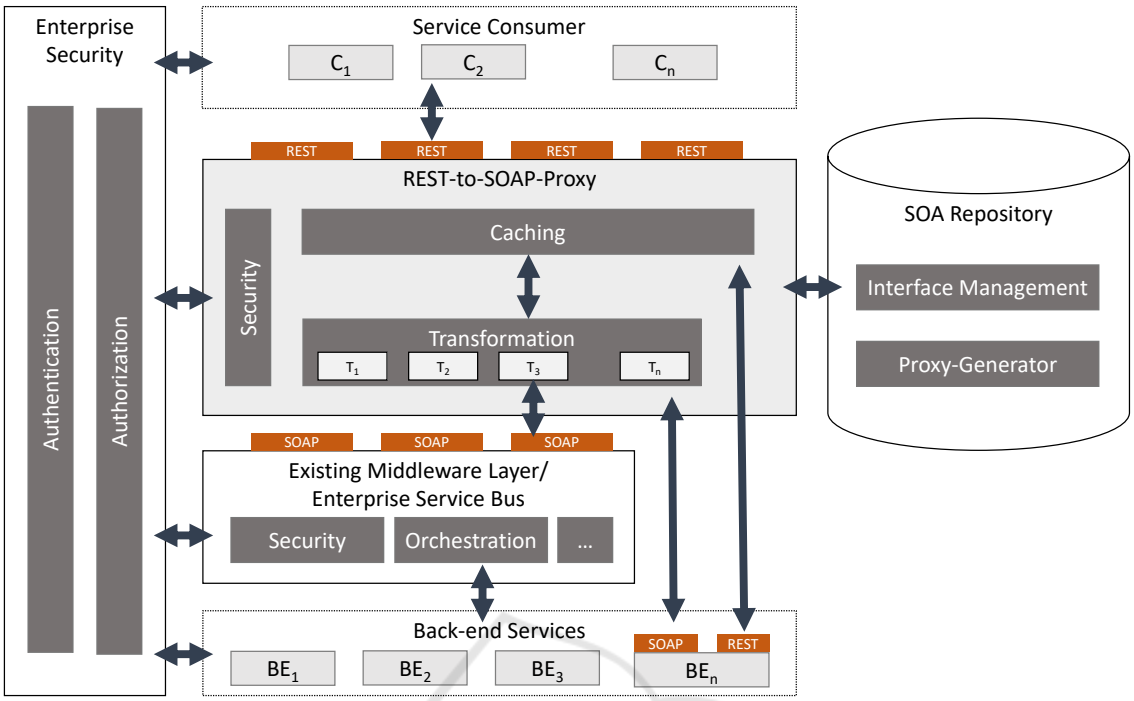

Figure 1: REST-to-SOAP Middleware Architecture.

data they have permission on and data is only transferred securely. On the other hand, it might be required that certain extremely sensitive data is never cached and only passed through the proxy. This of course leads to the already mentioned higher load on the back-end system, but in certain instances is worth the trade-off for higher security.

4.2.2 Types of Caches

There are different types of caches that can be considered for our architecture:

1. a central, user-independent cache,
2. a user-based cache and
3. a session-based cache.

The central cache can be compared to a standard HTTP cache that stores each requested resource once and subsequently delivers it independently from the identity of the requester. For services that are publicly accessible without any authentication this option would be the most obvious and efficient. In our case this solution is not viable, as services in an enterprise environment allow access to sensitive data and therefore require user authentication. A user-based cache stores resources separately for each user. This of course leads to increased storage requirements, because the same resource may be cached multiple times. To decrease the necessary storage space, the time a resource is cached can be reduced if a session-based cache is used. In this specialized version of a user-based cache, resources are removed from the cache as soon as a user's session ends, which might be significantly lower than the resources time-to-live.

4.2.3 Position of the Cache

Each response from a back-end service is translated with the help of the transformation component into a REST-compatible JSON format corresponding to the defined data format. This allows for two positions of the cache. It can either be located before or after the transformation component, which also influences how the cache operates.

If located before the transformation, the cached data needs to be transformed again for each subsequent request, increasing the load on the proxy. To prevent this, the cache can instead be placed after the transformation, having the data already transformed for subsequent requests. On the other hand, caching the data before the transformation would allow to reuse it for different types of requests, keeping the cache size small. Because of the already discussed security requirements, different requests have to be answered at least once by a back-end service to guarantee that the requester indeed has the necessary authorization. The use of cached responses is therefore not possible, which requires us to position the cache after the transformation in our architecture. There

may of course be additional HTTP caches between service consumer and the REST API. Those are part of the underlying communication and network infrastructure and therefore are not part of our architecture.

4.3 HATEOAS and Hypermedia Controls

One of the key concepts of a REST API is that it is self-describing, i.e. it informs the user how the API can be used and which operations are available on a resource and what URIs have to be queried. This concept is described by the term *Hypermedia as the Engine of Application State* (HATEOAS) (Fielding, 2008). Therefore it is possible to use a REST APIs without any prior knowledge about the interfaces and possible operations as long as an entry URI is known. Within enterprise environments this behavior is often undesired, because stability, and therefore clearly defined interfaces, are preferred. Still, these aspects have to be taken into consideration.

Our R2SMA supports, at a conceptual level, HATEOAS and hypermedia controls. During generation of a REST proxy the operations of a conventional web service are matched onto resources that are then used by the REST proxy. This in theory allows to deliver URIs for subsequent requests and operations on a resource with each response.

4.4 Identity and Access Management

To secure access to the REST API proxies, the R2SMA needs to support enterprise security features. It does not itself authenticate users. Instead it propagates login information such as user name-password-combinations or single sign-on tokens to the back-end services. The back-end services then authorize the requests and return a result or in case authorization was not successful return an appropriate error message. Additionally, the R2SMA is also tied into enterprise security to validate single sign-on tokens, for example to access cached responses.

Conventional web services support a number of advanced security mechanisms like the Web Services Security specification [3] building on XML encryption and XML signatures. For the realization of the REST proxies it is necessary to build on security mechanisms that are supported by both REST APIs and conventional web services. The R2SMA relies on Security Assertion Markup Language (SAML[4]) tokens for that. Also feasible would be the use of HTTP basic authentication und propagation of the login information to the back-end service.

4.5 Interface Management

The SOA repository is also responsible for the management of interface description documents, for both REST APIs and conventional web services. For the description of REST interfaces the R2SMA relies on the OpenAPI standard (Open API Initiative, 2017). OpenAPI has become widely used in the last few years and is supported by large companies such as Microsoft, Google and Atlassian.

Description documents are provided by the repository. Users can access them to integrate the services into their applications. The repository also handles the interfaces' life cycle and versions. This for example includes notification of users about upcoming changes to interfaces

4.6 Proxy Generator

The proxy generator component located in the SOA repository is responsible for the creation of REST proxies and accompanying interface description documents. The repository also contains the web service description documents for all existing services. Based on these descriptions, a skeleton for a corresponding REST proxy is generated. This skeleton is then completed by a user who maps the web service operations onto REST resources and defines the appropriate HTTP methods. The data models defined in the WSDL file are then transformed into a JSON schema description. With the manually added mappings and the automatically created service skeleton and JSON schemas, a REST proxy component can be generated and deployed on the REST-to-SOAP-Proxy.

5 IMPLEMENTATION

The presented architecture provides a blueprint for the realization of a REST-to-SOAP Middleware. The specific selection of technologies is highly dependent on the IT ecosystem of a company. This section provides an overview over possible technologies that can be utilized for realization. It also presents a prototypical implementation of select components.

5.1 Implementation Technologies

For the realization of the proxy, including cache, transformation and security, off-the-shelf software products can be used. Application servers usually

[3] https://www.oasisopen.org/standards#wssv1.1.1
[4] https://tools.ietf.org/html/rfc7522

Figure 2: Prototypical user interface for proxy generation in the *SOA Governance Repository*.

already support caching and provide security functionalities which can be extended and used for our R2SMA. Relevant application server vendors are for example WSO2[5], Mulesoft[6] and IBM[7]. Transformation can be realized using Java libraries or XSLT style sheets. The transformation of messages between JSON and XML can be handled by existing libraries like XSLTJSON[8]. Approaches such as (Nogatz and Frühwirth, 2014) could also be used to convert XML to JSON schema.

As a repository we see our previously developed *SOA Governance Repository* (Königsberger and Mitschang, 2016) as an ideal fit. The repository serves as a one stop shop for all information about the state of an enterprise SOA and provides documentation about services and stakeholders. Of course, the functionality could be developed as a stand-alone tool. Possibly other off-the-shelf products could be used as a basis, for example the products discussed in Section 3, which already provide basic functionality for creating REST proxies.

5.2 Prototypical Implementation

We implemented prototypes for select R2SMA components and present them in the following section.

Proxy Generator. The *SOA Governance Repository* stores the service and interface descriptions of web services as RDF triples (Resource Description Framework[9]). This is accomplished by parsing the WSDL files of the services. The implementation of the *Proxy Generator* uses this information to create the skeletons for the REST proxies.

[5]https://www.wso2.com
[6]https://www.mulesoft.com
[7]https://www.ibm.com
[8]https://github.com/bramstein/xsltjson/
[9]https://www.w3.org/TR/rdf11-concepts/

Because of the already discussed paradigm differences between REST APIs (resource-centric) and conventional web services (operation-based), a manual step is necessary to create a proxy. In this step, a technical stakeholder has to map the operations onto resource URIs and define the corresponding HTTP methods. The base URI for the REST API is also defined in this step. Figure 2 shows the user interface of the *SOA Governance Repository* for the creation of REST proxies.

The system supports the user by providing suggestions for HTTP methods, based on the names of the operations. For example, the method *GET* is suggested if the operation name contains the keywords *get* or *retrieve*. This information is then also stored as triples in the *SOA Governance Repository* database and later used for the generation of a REST proxy. A proxy consists of a generated JAVA library that can be deployed on the application server. It exposes a REST API to the service consumers and utilizes an accompanying transformation to translate the incoming requests into SOAP and vice versa. Currently caching isn't implemented in the prototype but will be part of further development.

REST API Documentation. The information used for the generation of REST proxies is also used to create their interface description documents. The interfaces are described using the OpenAPI specification (Open API Initiative, 2017). The information necessary to create the description document is stored in the *SOA Governance Repository*. It is extracted from the parsed WSDL files as well as manually entered during the operation-to-resource mapping.

The *SOA Governance Repository* also manages so called Business Objects *plus* (BO^+), we previously presented in (Königsberger and Mitschang, 2017). These BO^+ describe the data models for service interfaces independently from any specific description

language. They too are stored as RDF triples and can be transformed into a XML schema definition as well as a REST-compatible JSON schema (Internet Engineering Task Force, 2017). The transformation from RDF triples to a JSON schema can be realized using tools such as owl2jsonschema[10].

6 USE CASE EXAMPLE

This section gives a short use case example on how a REST API proxy is generated from a web service and how the mapping from operations to resources is realized.

The following code listing describes a simple web service using WSDL 2.0. The service *MyHRPortal* contains four operations to manage employees. With these operations existing employees (one or many) can be retrieved, deleted or new ones can be created. The WSDL has been shortened and non-relevant parts have been omitted.

```
<?xml version="1.0" encoding="UTF-8"?>
<description [...] >
  <types>
    <xs:schema [...] >
      <xs:element name="employeeID"> ...
    [...]
    </xs:schema>
  </types>

  <interface name="HRPortal"> [...] <interface>

  <binding name="SoapBinding"
           interface="tns:HRPortal"
           type="...w3.org/ns/wsdl/soap"
           [...] >
    <operation ref="tns:retrieveEmployees" />
    <operation ref="tns:retrieveEmployee" />
    <operation ref="tns:createEmployee" />
    <operation ref="tns:removeEmployee" />
  </binding>

  <service name="MyHRPortal"
           interface="tns:HRPortal">
    <endpoint name="SoapBinding"
              binding="tns:SoapBinding"
              address=".../hr/soap/"/>
  </service>
</description>
```

This WSDL document is parsed by and stored in the R2SMA's repository component. From this information a REST proxy skeleton is created already containing most of the abstract contents of the parsed WSDL, such as service name, description and data types. Then, a service developer (or other technical stakeholder) would define the mapping from operations onto resources. Table 1 shows how the mapping might be defined by the developer for this specific use case. {id} is used as a placeholder. In a request it would hold a specific ID of an employee and the value is mapped to the property *employeeID* defined in the WSDL file.

Table 1: Web service operation to REST resource mapping.

Web Service Operation	REST Resource	HTTP Method
retrieveEmployees	/employee	GET
retrieveEmployee	/employee/{id}	GET
createEmployee	/employee	POST
removeEmployee	/employee/{id}	DELETE

A REST API proxy following this mapping and providing these resource endpoints can then be made available through the R2SMA for use by service consumers.

7 ASSESSMENT

The use case example presented in the previous section is straightforward and the mapping might even have been done automatically. Unfortunately, not all services can be mapped this easily. Especially if a service's operations have been defined in an unstructured way, a correct mapping following REST guidelines may not always be possible. In such cases lower maturity levels for an API may need to be accepted. The services could be changed to allow for an easier mapping onto REST structures, but that would go against the basic concept of our architecture to provide REST API access without modifying existing services.

The presented architecture itself, with its abstraction of back-end services through the Rest-to-SOAP proxy middleware (R_2) and the manual definition of resources based on the operations defined in the WSDL, fulfills level 2 of the Richardson Maturity Model (Fowler, 2010) (R_3). To achieve the highest level 3, the support of full HATEOAS would be necessary. This paper lays the groundwork for that, but for full-fledged HATEOAS, it would be necessary to further link the operations to define related actions on certain resources. Here, further research is necessary to be able to achieve this automatically.

All components of the architecture can also be deployed utilizing load balancers or fail-over system to achieve enterprise-grade reliability (R_3).

The architecture adds an additional abstraction layer, which also has an effect on overall performance

[10]https://github.com/redaktor/owl2jsonschema.js

and response times. Due to the additional transformation, slightly negative effects on performance might be possible, although the transformations themselves are not computation-intensive. The newly introduced cache of the R2SMA might even lead to an improvement in performance and response times, as certain repeating requests that have been answered by the back-end services in the past now can be retrieved directly from the cache.

8 SUMMARY AND OUTLOOK

In summary, the REST-to-SOAP Middleware Architecture presented in this paper allows the continued usage of an existing web service infrastructure for new and more lightweight use cases without the need to implement massive changes for these existing services (R_1). The architecture also allows to open up new areas of business and for example make existing services available to external developers on a per-use basis. The cache that is part of the architecture allows to keep the load on the back-end systems as minimal as possible while at the same time still providing the necessary level of security and supporting RESTful features such as the partial retrieval of lists (R_4). This means that there is only minimal, if any, investment into more powerful hardware necessary to handle the additional load. The integration of a SOA repository allows for the management of both REST APIs as well as conventional web services in one place (R_5) and provides the information basis for a semi-automatic generation of REST APIs.

Future work will include the prototypical development of the caching component as well as an experimental evaluation of the architecture within an enterprise context to determine its feasibility in practice.

ACKNOWLEDGEMENTS

This work is partially funded by the BMWi project IC4F (01MA17008G). The authors would also like to thank H. Berner for valuable discussions and insights from an enterprise perspective.

REFERENCES

Agrawal, N. (2015). HowTo – REST API proxy to SOAP webservice. https://blogs.mulesoft.com/dev/howto/rest-api-proxy-to-soap-webservice.

Apigee (2014). New Apigee Wizard Converts APIs from SOAP to REST in Minutes. https://apigee.com/about/press-release/new-apigee-wizard-converts-apis-soap-rest-minutes.

Bossert, O., Ip, C., and Laartz, J. (2014). A two-speed IT architecture for the digital enterprise. http://www.mckinsey.com/business-functions/business-technology/our-insights/a-two-speed-it-architecture-for-the-digital-enterprise.

de Giorgio, T., Ripa, G., and Zuccalà, M. (2010). An Approach to Enable Replacement of SOAP Services and REST Services in Lightweight Processes. In *Current Trends in Web Engineering*. Springer.

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

Fielding, R. T. (2008). REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

Fowler, M. (2010). Richardson Maturity Model: Steps Toward the Glory of REST. https://martinfowler.com/articles/richardsonMaturityModel.html.

Internet Engineering Task Force (2017). JSON Schema. https://tools.ietf.org/html/draft-handrews-json-schema-00.

Königsberger, J. and Mitschang, B. (2016). A Semantically-enabled SOA Governance Repository. In *Proceedings of the 2016 IEEE 17th International Conference on Information Reuse and Integration*. IEEE Computer Society.

Königsberger, J. and Mitschang, B. (2017). Business Objects plus (BO+): An Approach to Enhance Service Reuse and Integration in Cross-Domain SOA Compounds. In *Proceedings of the 2017 IEEE 18th International Conference on Information Reuse and Integration*. IEEE Computer Society.

Nogatz, F. and Frühwirth, T. W. (2014). From XML Schema to JSON Schema: Translation with CHR. In *Proceedings of the Eleventh Workshop on Contraint Handling Rules*, volume abs/1406.2125.

Open API Initiative (2017). The OpenAPI Specification. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*. ACM.

Peng, Y. Y., Ma, S. P., and Lee, J. (2009). REST2SOAP: A framework to integrate SOAP services and RESTful services. In *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*.