Declarative Web Service Entities with Virtual Endpoints

Martin Grund  
Hasso-Plattner-Institute  
EAA Lab August-Bebel-Str. 88, 14439 Potsdam, Germany  
martin.grund@hpi.uni-potsdam.de

Jens Krueger  
jens.krueger@hpi.uni-potsdam.de

Dr. Alexander Zeier  
alexander.zeier@hpi.uni-potsdam.de

Abstract

The shape of entities used in web services are mainly defined with the idea of a specific business case in mind. As soon as any third party uses these web services with another perception of the process, the shape of the entities may not be optimal any longer. Our paper proposes a solution that involves the service consumer party by letting the service consumer re-define the entities. Two architectural approaches are shown that provide an unobtrusive solution towards dynamic shaped web service entities. Furthermore we discuss the principles of declarative web services and the usage of context oriented programming for our concept of virtual endpoints.

1. Introduction

One of the key challenges for designers of Service Oriented Architectures (SOA) is the granularity of the provided services. There are many different ways of how to design an interface for a single service, granularity being the main differentiator between the different approaches. But why is service granularity such an important issue? The answer to this question is simple: It does not only have a massive impact on the understandability and thus usability of the service, but it also significantly affects its performance and its suitability for the specific business or domain. In this paper we will address the granularity issue and propose a solution that eases the modeling, simplifies the runtime architecture and development process of web service for both parties participating in the service consumption process. The first part of this paper will give a short introduction into granularity of web services. The second part will describe the business scenario and how we came up with our idea by giving an example of a service consumption scenario and outlining the major performance and granularity problems here. In the third part we will propose our new idea of virtual endpoints and explain how to overcome the granularity problem for complex types by defining views on web service entities. The last part of this paper will show related work and give a conclusion and outlook regarding the topic of virtual endpoints.

2. Granularity Aspects

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.[10]

The service should be specified with an application or the user of the service in mind, while the service realization may be provided by a software package, e.g. an Enterprise Resource Planning (ERP) package, a special purpose built component, commercial off the shelf applications, or a legacy application. To a service client is irrelevant whether the services are provided by a fine grained suite of components, or a single monolithic ERP system. It is important that the developer who implements the service is aware of the services’ granularity so he can change parts of the implementation with the minimum disruption to other components, applications and services. The granularity of components should be the prime concern of the developer responsible for providing component implementations for services.[12] Regarding the component composition aspect we can distinguish between three major granularity aspects as stated by Dan Foody.[7]

1. Performance and size - means considering the
amount of round trips for a certain scenario as well as network latency or overall message size that affects the serialization and de-serialization process on the provider and on the consumer side.

2. **Transactionality and state** - defines the granularity regarding the transient state to be maintained between operations on the service provider side, the transactionality aspect regarding operations spanning over multiple web service calls and the number of operations performed within a single service call.

3. **Business suitability** - defines the granularity aspects concerning the actual business semantics of the process that is represented with the use of web services and the point of view of the service provider concerning the provided operations and entities.

While the aspects of transactionality, state and business suitability need to be solved on the server side and mainly without incorporation of the client or service consumer the performance and size issues should be addressed from both sides. Not only the service provider is responsible to provide adequate services but also the service consumer is responsible to invoke only those services interface reasonable for the business process in execution. In this paper we will address the granularity aspect of performance and size in a special way. Instead of assuming that only the server side needs to adapt to the client requests, we assume that the service consumer must pro-actively adapt to his requirements as well. We are aware that the proposed approach does not solve all granularity issues completely. From our point of view our paper shows the direction to proceed to allow better service consumption and achieve better performance for web service composition.

2.1. **Business Scenario**

During our research we worked on many different topics in the area of web service consumption. During the time we noticed typical problems that belong to the topic of understanding a specific business domain. Web services provided by enterprise systems are often aligned to the implementation provided by the vendor. The granularity of web services mainly reflects the behavior and foreseen composition in respect to the underlying business process. Yet we learned that one has to distinguish between different types of web services. On the one hand there exist **action based web services**. These web services execute a certain action or step in the business process when executed. The returned result set is relatively small and limited. An example for this kind of web services in the area of customer relationship management (CRM) is a web service that releases a sales order. On the other hand there exist **entity based web services**. These services are mainly used in the context of CRUD\(^1\) operations and especially in the case of reading operations produce a lot of output data.

\[\text{Figure 1. Different perception of a business object}\]

For the consumer trying to incorporate with those enterprise web services this discloses different obstacles. On the one hand the data is often distributed across different services or the complex type definition of the parameters and return values is not meaningful since the web services are often directly generated from technical modules. On the other hand the consumer does not has any influence on the granularity of the consumed entities.

An entity in this use-case defines any complex type representing a typical data type from the business domain of the consumed web service. As a result many different web service calls are executed subsequently to read the necessary data. Especially in the context of applications that have not been foreseen by the vendor of the enterprise services the granularity problem of entities becomes obvious. As seen in Figure 1 the understanding of a certain business object representing the entity later on used in the web service differs between the point of view of the service provider and the point of view of the service consumer. The different perception lets the service consumer build new applications, typically not provided by the service provider. Usually the data needed for these applications does not have the same shape as the data that was originally foreseen when consuming the web service. This is typically the case if an application acts as build on top of another application referring to this base application as a platform.

2.2. **Example**

During a research project we developed a front-end for the Order-To-Cash scenario provided with web service by...\(^1\)Create, Read, Update, Delete
an ERP system. For example an implemented view reads all open sales orders for a given customer so that the customer is able to browse through all sales orders and their main information without refreshing the page. When clicking on a single sales order the customer should be taken to another web page displaying the sales order and its details together with material details.

The following enumeration lists all functional requirements to realize the customers view.

F1 Find all open sales orders
F2.1 Read single sales order
F2.2 Read sales order items for sales order
F2.3 Read material details for a sales order item

The next step is to find the appropriate services to fulfill these requirements. This step becomes complex because often the service provide only a sub- or superset for the required functionality. The identified services for our use case are listed below.

S1 Find Open Sales Orders by Customer
S2.1 Read Sales Order by Id
S2.2 Read Material Basic Info

As one can see in Figure 2 this scenario has a huge drawback. The first service (S1) retrieves all open sales orders belonging to a single customer, but returns only the IDs of the sales orders. So displaying all open sales orders with a certain amount of detail means first executing the `retrieveAll` call and afterwards iterating over all sales orders and reading the full sales order from the system, even if the information to be displayed in the current view is only a small subset of the full sales order.

Of course it is possible to argue that read data F1 could be reused in F2.1. The disadvantage would be the need to implement caching mechanisms in a client that should have a fast and simple architecture. Furthermore the additional complexity is introduced only because of the fact that the web service does not provide the data in the right granularity for the web service consumer.

As a result the overall performance of the web service decreases because unnecessary data is first serialized, then transmitted over the wire and de-serialized. Not only the performance of the current web service call is affected, but as well the overall client application performance in a way that multiple service calls are needed to fetch data that originally would belong together. The critical point revealed in this very simple use case is that the service provider cannot foresee in which granularity the data provided is needed.

### 3. Virtual Endpoints

To overcome the problem of different data granularity we propose an addition to the today’s way of providing web services. Usually, services consumers first identify the data their application needs. Then they look for a service that provides a particular piece of data. Afterwards they learn the services programming interface, i.e. its input and output data types, possible preconditions and side effects. If the service requires input data that is not available within the application state and context, they will have to look for additional services that provide missing data.

As stated in 2.2 the granularity of a web service data type can become crucial for its performance. The question is now how it is possible to accomplish the goal of providing web services with different granularity for the used web service entities, but to provide a stable and reliable service interface.

The basic architecture for our idea is the following: Since the usual service consumer usually performs the service orchestration by hand, we propose to offer a web service entity design time builder together with virtual endpoints. The foundation for each web service is the full data type provided by the underlying application server. Now
as a next step the consumer is capable of either using the web service as-is directly from the vendor and to take into account that the granularity of the data types used for the provided CRUD operations may not reflect the information that the service consumer needs to accomplish its tasks. If the service consumer does not want to use the full interface he can use a design time tool to create his own view on the underlying data entity.

The data stored in such an entity is a superset of the data actually needed by the service consumer. Each use case the business entity is used in provides only a view on the business entity. Formerly the web service data types are shaped by this use case. Unfortunately the use case is specified by the provider of the web service and defines not always the same use case of the web service consumer. With a web service entity design time the consumer is capable of shaping the web service data types as they are needed in the consumers application or context. As a result the overall time needed for serialization, transmission, de-serialization is reduced and the overall message size can be dramatically reduced. Furthermore subsequent service calls to first fetch an identifier and afterwards reading the details are no longer needed. An example for the view definition can be seen in Figure 3 - here instead of using the full SalesOrder entity with all sales order items and the related information on the header document, for the same method findById a different object is returned. But the different view is not different from a semantic point of view but only from a data description point of view.

Since now every service consumer can define his own view on the result of a service the question is now how the architecture for the service runtime should look like. Here again we tried to follow a declarative approach. For each view on the result set of the web service a so-called virtual endpoint is generated. It is crucial that the virtual endpoint does not provide own implementation logic, because it only reflects a view on the underlying data. Still even when using the virtual endpoints, the overall application logic and context is provided by the base service implementation.

### 3.1 Wrapped Virtual Endpoints

The first idea of an implementation for such a runtime would be as depicted in Figure 3.1. As one can see the application module exposes the web service in a standard way and does not need to be changed to adapt the principle of virtual endpoints. An application module is hereby not limited to a real model but is the placeholder for any object, module or unit that can be exposed as a web service. The service interface provided by the application module describes the operations available and the data types used by this service interface. When the service consumer defines a new view for the data types used by the web service this mapping is stored and out of this mapping the virtual endpoint is generated as described in 3.3. In case the web service is called the method call is handed over to the application module just as it would be a normal call. But instead of directly serializing the return value, as it would be done by the original service runtime, the return value is inspected and serialized according to the view specification defined by the service consumer. In the example mentioned in 2.2 and shown in Figure 3 a view could remove associations and attributes defined on the base return value. The resulting return value respects the schema definition for the complex type defined by the virtual endpoint but has a unique advantage. Since it does not reflect the full return value, the time needed for serialization is shorter and the overall message size that needs to be transmitted to the consumer side is reduced. To summarize, the virtual endpoint acts as wrapper for the original service implementation.

![Figure 3. Web service entity view](image)

![Figure 4. Simple Virtual Endpoints](image)
3.2 Declarative Virtual Endpoints

The second approach for the virtual endpoint is based on the idea that the web service implementation should benefit from the knowledge about the newly shaped web service entity. Database query language like SQL, XQuery etc are heavily oriented towards a declarative approach. Instead of defining how to fetch data, these data sublanguages describe what to fetch with which conditions. Web service interfaces describe a set of operations for a specific object/module. Since our approach of virtual endpoints follows the idea of letting the service consumer describe - declare - which data the consumer wants to fetch. The virtual end point runtime should be able to use this additional knowledge in a way so that the application module implementing the exposed web service benefits from this knowledge as well.

The first step towards declarative virtual endpoints is to analyze what happens when for example a sales order is read like described in 2.2. From the understanding of the service interface the operation `FindOpenSalesOrdersByCustomer` reads all sales orders fulfilling a certain condition. Our interpretation of this typical data related operation could be an SQL statement like described in Listing 1.

```
Listing 1. Virtual SQL statement for retrieving all sales orders

SELECT totalValue, totalTaxValue, status, buyerParty, deliveryParty
FROM sales_orders
WHERE status = 'open' AND buyerParty = 1
```

Even if the service consumer only needs information about the total value of the sales order all information associated with the sales order is read from the persistence layer and leads to a longer operational time of the method. If one would optimize this sample SQL statement one would remove all attributes from the projection which are not directly used in the following context. So now we need to ask the question why should the web service runtime not follow this recommendation if it would be aware of the expected return value and its defined shape?

In the first step the web service consumer already analyzed what data the consumer requires and was able to model this using the virtual endpoint design time. The view that the web service consumer defines using the modeling tool always defines a subset of the overall available data model. Assuming that the subset definition somehow reflects the data model on the persistence layer, the best solution would be to delegate this view information to the persistence layer.

Following the example the service consumer defines a subset of the original virtual SQL statement in terms of the elements selected in the projection. The example SQL statement can be described as shown in Listing 2.

```
Listing 2. Virtual SQL statement for retrieving all sales orders

SELECT totalValue
FROM sales_orders
WHERE status = 'open' AND buyerParty = 1
```

The web service runtime capable of understanding the concepts behind virtual endpoints sends this information back to the persistence part of the application module defining the web service. As a result only the necessary fields are fetched from the underlying data model. This results in a faster database query on the one hand and on the other hand the result is a return value that is way smaller from the complexity point of view than the original. The process serializing the return value into the corresponding XML format takes less time and the overall message size for this method call is smaller than for the original message. The advantages of this solution are obvious. Through the whole message request/response life cycle this solution proposes optimizations. Of course the optimizations for a single message do not seem to be huge but in respect to the overall amount of messages sent through a web service this solution reduces the overall load of the server and allows more concurrent connections since serialization of message from the return values to XML is a typical CPU bound operation. The freed resources can now be used for new incoming connections. Furthermore if less time is used to convert messages the overall round-trip times for request/response cycles decrease and therefore the web service call latency decreases.
3.3. Virtual Endpoint Definition

Once the web service runtime for virtual endpoints is available the key question to follow is how to enable the web service consumer to model the view on the entities used by the web services. Besides the fact that a tool with a graphical user interface would help a lot, we identified two approaches that are worth to follow.

One approach is a per interface definition of entities. For each interface that is accessed by the service consumer, he is able to redefine the entity used returned by the web service. This allows a very high granularity for the consumer, but increases the complexity of the service consumption process as well. To avoid unnecessary complexity, we propose that for each service a standard should be provided and only in the case, the model shaping does not fulfill the requirements of the consumer, the service consumer is able to reshape the entity provided by the web service in a way the service consumer requires. The advantage of this solution is that the service consumer has full flexibility to shape the entities provided by the web service in a way that is most suitable and allows him to reach the best performance for the desired service composition.

The second approach is to enable the service consumer to shape the entities provided by the service provider not on a per interface basis, but for the overall scenario. In case a single entity is used in more than one web service, the service consumer is now able to shape and redefine this model for all web services exporting this entity as a complex type. The advantage of this scenario is that it reduces the necessary administrative overhead for shaping the entities. The disadvantage by using this approach could be that the service entities are always shaped broader than they should, because the reuse factor is higher and the entities are shaped as a compromise between all services.

3.4. Proposed Implementation Strategy

For an implementation we assume that the server providing the basic container implementation would be an Java Enterprise Edition (JEE) 5 server implementation. With Java 5 annotations have been introduced in the language to allow a more declarative way of defining application semantic. In the enterprise application server runtimes annotations are a widely used mean to support definition of entities or web services. In a typical JEE 5 server for defining entities the concept of entity beans is used. The advantage of this concept is that the developer only needs to define the actual business logic. The framework creating, activating and persisting the entity bean will provide the necessary methods for data retrieval and updates. The API for this abstraction is called JPA (Java Persistence API) and is furthermore defined in the specification for Enterprise JavaBeans 3.0 in JSR 220[2]. Here the annotations introduced with Java 5 are used to declare the relation to the underlying data structures. Furthermore annotations allow defining associations with their cardinality and additional configuration parameters for persistent attributes.

Furthermore typically the Java API for XML-Based Web Services (JAX-WS) 2.0 is used to declare which classes are exposed as web services at runtime and which methods in these classes are exposed and under which circumstances. The specification for this API is defined in Java Specification Request 224[8].

Both frameworks provide annotations to make development of web services and persistent objects easier and to integrate the necessary configuration not in external XML files but directly where the classes are defined. As a result the necessary code is smaller and easier to maintain. Since almost all information that is required for configuration of either persistence or web service exposure can be accessed via introspection the idea is now to enable this combination of frameworks to work with virtual interfaces. The only thing that is not provided by these two frameworks is the actual link between the persistence and the web service API. Both frameworks operate totally independently from each other. Our implementation idea proposes the missing link. The process during a web service request response cycle would be as depicted in Figure 3.4 - a simplified activity diagram of the request/response scenario. As one can see it is obvious that most of the time the meta description for the current view on the data model is carried with the execution of the method call.

![Figure 6. Activity diagram for request/response cycle](image-url)

To avoid unnecessary code duplication or polluting the
either by inserting

actual application module code with configuration specific annotations we propose to use a context oriented programming model to build a virtual endpoint runtime system. A common goal for programming frameworks for context-aware computing is to enable programmers to easily develop and deploy context-aware applications. Programmers can focus on modeling and using context information and functionality specific for their application while relying on a basic infrastructure to handle the actual management and distribution of this information.[3] Currently, the only way to introduce context-dependent behavior into a program is either by inserting if statements everywhere that check for context in which a program is running but this would violate the principles of object-oriented programming by using polymorphic behavior. The other approach would be to factor out the context dependent part into separate objects that can be substituted according to the context. Both approaches lead to complicated code that is hard to comprehend and to maintain. With context oriented programming this can be avoided and is discussed in [4, 5].

When using context oriented programming we propose the following structure for the web service runtime: Semantically each virtual endpoint defines a view on the return value for a certain method in a certain web service interface. With the design time of the virtual endpoint, for each unique view on the defined complex types of a web service a specific layer is generated. The code for this layer is generated and stored with the engine. When the virtual endpoint is accessed with a request from the outside the correct context is calculated. In terms of context oriented programming one or more layers can be activated now. Since all views are disjoint only one layer is activated at a time, the base module code is hereby not counted as an additional layer. When the original code passes the borders of the persistence framework the framework will execute different database statements based on the currently activated layer. The return value is then passed back to the application module that handles the return value just as it would do with any other value. The serializing framework is again context aware and through the currently activated layer serializes the necessary fields and returns the response back to the requester.

This procedure allows a very easy separation of concern without modifying the original application module since all changes are made outside of the module. Changes need only to be made to the server runtime and are mostly unobtrusive regarding the actual implementations. Regarding the performance, of course, there is an overhead due to the use of a context-oriented framework but since the code for the actual layer is generated only a small part of the actual implementation is really dynamic.

To summarize, we propose an architecture that is as less obtrusive as possible and uses a context-oriented approach to be integrated in a typical JEE 5 engine to enable the use of virtual endpoints.

3.5. Service life cycle and administration

The idea of virtual endpoints could increase the complexity of the web service management, administration and life cycle. We assume that for the basic web service that acts as a template for the virtual endpoints exist a plan how to manage the web service life cycle, security etc. The concept of virtual endpoints only provides an addition to the already existing concepts. For us it seems to be the best idea to adapt the same mechanisms for the virtual endpoints as well so that each virtual endpoint behaves like its template. Security constraints that are applied to the template web service should be applied to the generated virtual endpoints as well. As a result the virtual endpoints have the same access rights as the template. The virtual endpoints should not provide any new layer of security or layer of information hiding but a tool to support the web service consumer. Thus we propose that the same security restrictions are applied on the parent template web service as on the virtual endpoint. In case that the underlying web service entity changes, the web service runtime needs to check if this affects generated virtual endpoints. If this is the case, when for example a data type was removed from the overall data object, the generated virtual endpoint should be marked as invalid. Furthermore an idea is to support web service versioning as stated in [9] to keep track of different versions of the parent template web service and of the generated endpoints on the other hand.

Furthermore the dynamic shaping of web service entities allows an easier reuse of web service packages since the consumption is no longer defined through the granularity of the web service interface but through the feature of the interface. This means instead of providing many different interfaces to read an object in different levels of granularity, our solution allows to reuse the interface and delegate the correct shaping to the service consumer.

4. Related Work

4.1. REST

REST is an architectural style that Roy T. Fielding first defined in his doctoral thesis[6]. REST specifies several architectural constraints intended to enhance performance, scalability and resource abstractions within distributed hypermedia systems. One of these is the uniform interface constraint, which means that all resources present the same interface to clients. Another is statelessness in which servers keep no state on the client’s behalf. Resources and representations are also key parts of REST. REST treats
representations as the way applications navigate distributed hypermedia systems; similarly, applications normally access SOA services via some sort of distributed network handle.[14]. The interesting point regarding the topic discussed in this paper is the resource and its representation. As already stated most of the time entities or resources are read from distributed systems. The URI describing a specific resource can as well be seen as an query language allowing to specify not only a single parameter but providing an expressive declarative query language.[13]

4.2. Context Oriented Programming

Contextual information is playing an increasingly important role for applications and services ranging from those that are location-based to those that are situation-based or even deeply personalized. When introducing a context oriented framework to an application server the opportunities are not only limited to the field of web services. One use case could be runtime adaption in order to deal with a host of unanticipated changes. Monitoring, tracing and profiling during system runtime are common examples. Changes in the problem domain or the underlying technology may make it necessary to inspect an already deployed software system.[1, 4, 5, 11].

5. Conclusion and Future Research

The idea as stated in this paper shows an urgent need for a more declarative way to express the web service consumption. This declarative approach should not only affect the implementation on the server side, but as well allow the client runtime to benefit from the declarative approach. The success of SQL, XQuery and other declarative query languages shows that at least for entity models it is very useful to be able to express in a declarative way which information to retrieve. Furthermore if the web service runtime supports such an declarative approach the web service definition does not describe the strategy on how to fetch the data from the underlying system, but only what to fetch. If this advantage is handed over to the application server the application server starts to adapt to the behavior of the web services and optimize the requested queries. Of course the proposal is mainly designed for entity based web services and affects typical action based web services only a little. But in typical ERP scenarios most of the time data is read and this is the point where most of the time is spent.

We think that future research in the area of services should be centered in the area of declarative query languages. Again the success of SQL shows that a query language is a very good abstraction between the actual persistence implementation and the consumer of data. Still the advantages of web services are obvious because the automated code generation on the client side and the strong typing approach ease the development of service clients. Both solution - the dynamic declarative and the static web service one - have their advantages. The combination of strong typed method invocations and dynamic data retrieval will allow future application servers to benefit from the knowledge the web service consumer has and to increase the overall application performance.

6 Acknowledgements

We want to thank Cafer Tosun, Anja Bog, Jan Schaffner and Matthieu-Patrick Schapranow.

References