RESTify: From RPCs to RESTful HTTP Design

Jakob Strauch
mobile media & communication lab
FH Aachen University of Applied Sciences
strauch@fh-aachen.de

Silvia Schreier
Chair of Data Processing Technology
University of Hagen
silvia.schreier@fernuni-hagen.de

ABSTRACT

Starting with RESTful design is a difficult task – even more if the designer has a RPC or object-oriented background. To support the adaption from RPC- to REST-oriented thinking, we propose RESTify, a straightforward procedure model to redesign a RPC-oriented interface into a hypermedia-enabled REST interface. RESTfiy uses a WSDL document of an existing SOAP service and consists of three iterations. The result of each iteration is an enhanced version of the preceding one concerning the REST constraints and is meant to be implemented as a HTTP service. Beside the technical result of the process and the design of a RESTful interface, the developer becomes acquainted to the main elements of a RESTful design, the constraints and their application. The results of the evaluation, using a prototypical web application and public SOAP services, are promising.

1. INTRODUCTION

The biggest challenge in migrating a RPC-based API to a RESTful design is the concept mismatch. Whereas RPC interfaces concentrate on meaningful operations, a resource-centric design requires a different view of the domain. In some public web APIs the RPCs remain, e. g., in terms of so called RPC URI-Tunneling. APIs using URI-Tunneling "expose resources but operations are tunneled through action parameters in URIs." [1]. Occasionally, there are 1:1 matches from a SOAP variant to its HTTP-based pendant.

Best practices for designing REST interfaces can be found but no general defined procedure that describes a transformation from a RPC-based API to a resource-oriented one. To deal with this issue, we propose an iterative procedure consisting of multiple iterations, each ending in a "more" RESTful design than the latter. The iterations are derived from different observations and (anti-)patterns.

Beside the procedure model and its technical implementation, this work focuses on helping developers with RPC background to understand REST and RESTful HTTP. The idea is to guide the developer using an example that starts from a real world WSDL document. Every iteration outlines some REST concepts to encourage the designer to improve the intermediate results in terms of RESTful HTTP. Furthermore, RESTify tries to raise the designers awareness of best practices as well as anti patterns – and how to resolve bad design choices in terms of REST.

After discussing related work in the next section, the iterations of the RESTify procedure model and its prototypical implementations are described in Section 3. The conclusions and the future work are presented in Section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WS-REST 2012, April 2012; Lyon, France
Copyright 2012 ACM 978-1-4503-1190-8/12/04 ...$10.00.

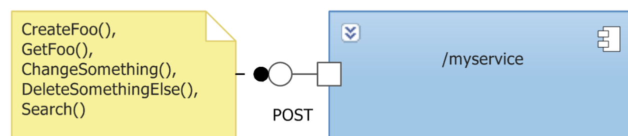

Figure 1: Example of an initial resource

2. RELATED WORK

The "Richardson Maturity Model" (RMM) [6] describes three maturity levels of Web APIs. The RMM is controversial [3] because it implicates a certain judgment of an API solely based on a fixed set of quality attributes. Despite the controversy, the RMM helps to judge an API design, but it lacks of detailed steps to achieve the ultimate REST level. Another model is the "Classification of HTTP Web APIs" (CoHA) [1], which evaluates common practices in public HTTP-based APIs. It describes the benefits and drawbacks of the different design techniques. The description of a procedure to reach for the highest classification is missing as well. Webber et al. [17] explain REST design from scratch, and Allamaraju [2] offers a collection of recipes for the designing RESTful services.

An approach to overcome the RPC-to-Resource gap is the protocol adapter StoRHm [9] but only simple CRUD[1] services can be mapped straight-forward. Liu et al. [12] introduce a process to re-engineer a legacy system to a RESTful interface, but some REST constraints (e. g., hypermedia) were disregarded. Adding hypermedia as an afterthought is suggested by Liskin et al. [11] but not for SOAP services. Laitkorpi et al. [10] present a model-driven approach for designing RESTful services starting from a functional specification and not existing RPC services. A model-driven process that identifies resources based on legacy service descrip-

[1]Create, Read, Update, Delete

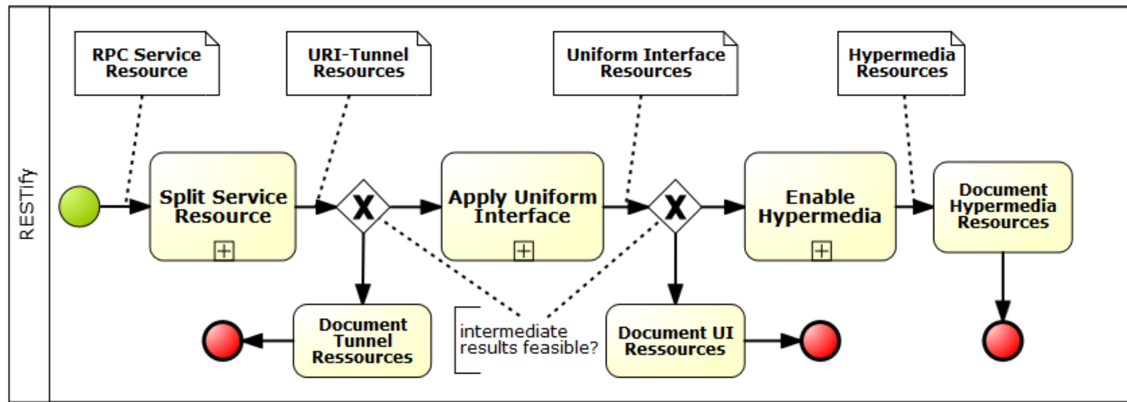

Figure 2: BPMN diagram of the procedure model

tions is described by Athanasopoulos and Kontogiannis [4], but they leave out media types and hyperlinks. Starting as an interface designer with SOAP/RPC background, there is no step-by-step method to transform an existing RPC-based design to a resource-oriented design while preserving the transformation process transparent.

Hence, we present RESTify, an iterative procedure model with semi-automated tasks to enable an interface designer to redesign an existing RPC interface description to a resource-oriented interface description.

3. RESTIFY PROCEDURE MODEL

Due to the RMM controversial, one main design goal of RESTify was to create an iterative procedure model, where every iteration respects more and more REST constraints, identifies additional resources, and highlights different system quality attributes. The first iteration focuses on the addressability and identification of resources, as well as the recognition of the origin SOAP operations. The second iteration focuses on self-descriptiveness, cacheability, and performance. The third iteration finally addresses integration, hypermedia, and thus loose coupling.

The intermediary results of every iteration are influenced by the described classifications. Results of the first iteration can be classified with the CoHa “RPC URI-Tunneling” level. The second and third iteration produce results at RMM level two and three. Another design goal is to enable an automated processing of simple tasks to support the redesign. Starting with a given WSDL (syntax) and the designer's knowledge of the SOAP service functionality (semantics), every iteration leads to an interface description, which will be revised to a more RESTful design.

Figure 2 depicts an overview of the iterations of the procedure model. To evaluate the procedure model, a prototypical web application has been developed. The procedure itself is implemented by a separate REST API. Every step in the procedure model can be found as a task or process resource in this API. The process resources define links to finished, current and next tasks. The media type is `application/hal+xml` [8]. The design of processes and tasks is inspired by jOpera [13]. The web application itself acts as a client for this REST API.

The prototype was designed with a pedagogical aspect: The user is guided by introductory steps, which describe REST concepts and the upcoming task. Some tasks are semi-automatic and provide default values for design decisions to be made.

```
− <portType name="AWSMechanicalTurkRequesterPortType">
   + <operation name="CreateHIT"></operation>
   + <operation name="DisposeHIT"></operation>
   + <operation name="DisableHIT"></operation>
   + <operation name="GetHIT"></operation>
   + <operation name="SetHITAsReviewing"></operation>
   + <operation name="ExtendHIT"></operation>
   + <operation name="ForceExpireHIT"></operation>
   + <operation name="SearchHITs"></operation>
```

Figure 3: Excerpt of the example WSDL document

The REST API provides syntactical analysis with the help of WordNet[2] which is a lexical database for the English language that groups words into synonym sets. RESTify uses WordNet to suggest word categories (nouns, verbs, other). Furthermore, RESTify utilizes known algorithms for word stemming [14] or pluralization. Currently, only simple services are supported. This means, RESTify does neither support additional WS-* protocols nor services based on specific design patterns like the command pattern [5].

As an example, parts of the Amazon Mechanical Turk WSDL[3] are used. Mechanical Turk[4] is a crowd sourcing market place to delegate complex tasks (e. g., image recognition) to humans.

A requester can upload Human Intelligent Tasks (HITs), which are then fulfilled by the workers. The workers can choose from a wide range of HITs and get paid if the requester is satisfied with the result. A HIT is the main domain concept of the service, supported by numerous operations. We only present a small excerpt for demonstration purposes (see Figure 3). The WSDL describes 39 operations with 106 complex data types and covers many aspects in RESTify.

RESTify uses URI Templates [7] to illustrate the resource

[2]`http://wordnet.princeton.edu/`
[3]`http://mechanicalturk.amazonaws.com/AWSMechanicalTurk/2011-10-01/AWSMechanicalTurkRequester.wsdl`
[4]`https://www.mturk.com/mturk/welcome`

design. The proposed URI templates in the following sections are only examples and may vary.

3.1 Iteration I: Split Service Resource

The objective of the first iteration (see Figure 4) is to split the single SOAP resource, as illustrated in Figure 1, into multiple URI tunnel resources. RPC URI-Tunneling is a widely used REST anti-pattern, which is described by CoHA as the successive level after WS-*. Although many REST constraints are violated, it can be found in many web APIs (e. g., flickr[5]).

This iteration is the first step toward identification of resources, which is the first of the uniform interface constraints. It uses syntactical conversions to achieve its goal and is based on the observation, that public APIs in the world wide web usually have descriptive identifiers. This fact can be utilized to define new resource URIs while preserving the transparency of the procedure model and its results for the designer.

3.1.1 Resolve Ambiguities

The first task is to resolve any syntactic and semantic ambiguities. The latter ones can be resolved by understanding the service documentation, e. g., some operations can be renamed to reflect their intention better. An operation named *AddEntry* could be renamed to *AddEntryToMyNewsFeed*. To support automatic processing, names are normalized by applying CamelCase. In few cases the letter casing needs to be corrected (e. g., *RegisterHITType*, instead of *RegisterHITtype*).

3.1.2 Tag Word Compounds

In this task, operation names are split into single words or expressions by tagging word compounds explicitly. Due to the normalized operation names (CamelCase), spaces can be introduced to split words. Brackets are used to tag the syntactical building blocks for the first interface design.

- *GetQualificationsForQualificationType*
- *Get Qualifications For* {*Qualification Type*}

In most cases, word compounds consists of nouns and fill words. But in rare cases two verbs can form a compound, too. The tagging task is restricted by one rule only. Multiple, single verbs are not allowed and usually not used in operation names. This is important to clearly identify the activities in the next task.

3.1.3 Extract Entity and Activities

Based on the observation of the methods of nine different public available WSDLs, some patterns in the method naming were identified:

- Entity methods: methods, which lack in an activity (verb) usually are meant to get data about an entity.
- Solely activity methods: a minor set of methods describe only activities like *Search*. These methods can be identified by a verb-only identifier.
- Simple activity methods: many methods describe activities with entities involved, like *RevokeQualification*.

[5]http://www.flickr.com/services/api/

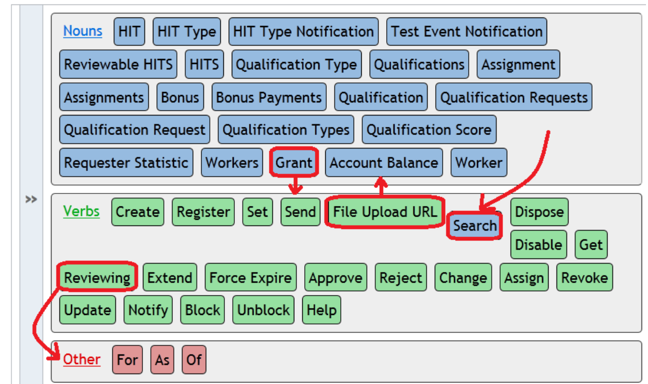

Figure 5: word type extraction (screenshot)

- CRUD methods: likewise simple activity methods but likely mappable to one of the HTTP Verbs. Examples: *CreateHIT* or *UpdateQualificationType*.
- Complex activity methods: method names consisting of more than one entity, maybe one verb, and zero or more filler indicate complex activities from a resource-oriented point of view (e. g., *MoveItemToCategory*).

Using these observations and preparations, nouns (entities or entity types) and verbs (activities) can be extracted.

Other word types refer either to the entity (e. g., *ReviewableHITS*) or the activity (e. g., *SearchExtended*). In the previous task, the designer could decide, e. g., whether *ReviewableHITS* is tagged as a word compound or two single words. Depending on such decisions, this results in more or less resources in the later tasks.

Although multiple verbs in one method are not allowed, in few cases auxiliary verbs are involved, e. g., *ForceExpireHIT*. This can be resolved by tag the verbs as compound ({*Force Expire*} *HIT*).

For the mechanical turk examples 22 entities and 21 activities can be extracted from the 39 methods (see Figure 5). With this categorization at hand, the SOAP interface can be split up.

3.1.4 Check Operation Naming Pattern

By highlighting the extracted word types with different colors in all operation names, usually a naming pattern can be observed. The pattern reflects the service-specific or designer-specific naming style, like *GetSomething*, *JustSomething*, or even *SomethingGet*.

Highlighted operation names, which differ from this pattern may indicate on the one hand a mixed style (in rare cases). On the other hand it may indicate a mistake or ambiguous tagging and/or classification.

In this task, the naming pattern should be checked, if it is reasonable and feasible. If an operation name contains two nouns, but no verb, a closer look could reveal a word miscategorization or a homonym[6]. In this cases, one can jump back to a prior task (see 5). If multiple nouns are encoded in a single operation name, a *designated entity* has to be chosen here.

[6]a word with multiple semantics, e. g., *list*, which can be a noun or a verb depending on the context

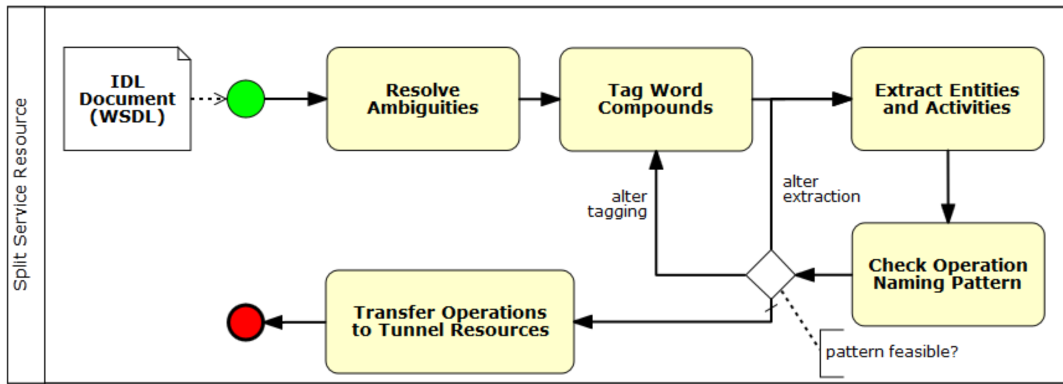

Figure 4: BPMN diagram of iteration I

3.1.5 *Transfer Operations to Tunnel Resources*

The last task in this iteration leads to a first interface description toward a RESTful design. Expressive URIs are used to name the resources. Therefore, two simple URI Templates [7] are used to reflect the resources origin:

- `/{designated-entity}?action={activity}`
- `/{activity}`

The first template will be applied to almost every case, while the second template was defined for operations with one verb only, e. g., *Search*. If the activity is not explicitly encoded in the operations name, the activity "get" will be assumed. The underlying URI structure was chosen to illustrate the RPC URI-tunneling anti-pattern. Hence, a SOAP method named *GetReviewableHITS* can be transferred into a URI like `/reviewable-hits?action=get`. The prototype supports this task completely, so that manual inputs are not necessary.

3.1.6 *Results*

As a result of this iteration, multiple resources are introduced by splitting the single SOAP resource into multiple parts according to its methods (see Figure 6). Some of them can be interpreted as one URI tunnel with multiple valid action values and the same designated entity type. Because every SOAP operation is transferred into exact one resource, the representation formats can be extracted from the WSDL itself and are set to the generic media type application/xml. The resources accept only HTTP POST.

In the mechanical turk example 39 resources can be defined (one for each SOAP method). Seven designated entities support multiple action values in 23 affected resources. Each designated entity shares the same URI Template with predefined action parameters. To pick up the example operations of the WSDL excerpt, the resulting *HIT* resource set (`/hit?action={activity}`) offers `create`, `dispose`, `disable`, `get`, `extend` and `force-expire` as valid actions. Each of the 16 remaining resources has only one valid action.

This first iteration introduces additional resources, which is just a little more RESTful in terms of the RMM. The results can be used to implement simple HTTP wrapper for the SOAP method pendants. The resulting resources are simple handlers for *plain old xml* payload, and many REST constraints are still violated, e. g., self-descriptive messages, which is respected in the next iteration.

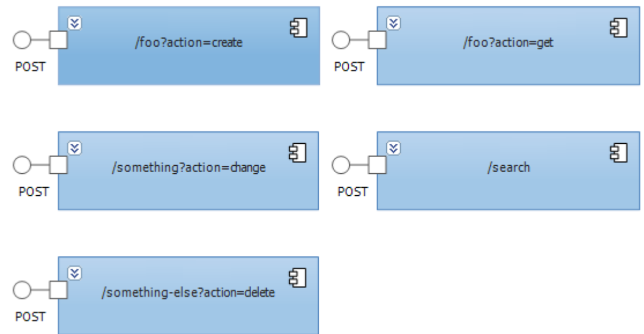

Figure 6: Example of resources after iteration I

The results of the first iteration are resources following the RPC URI-Template anti-pattern.

3.2 Iteration II: Apply Uniform Interface

In the second iteration (see Figure 7), the overloading of POST is replaced by following the uniform interface constraint. The key problem is that the resources from the first iteration represent in fact multiple resources, although each should be addressable, to follow the constraints of resource identification and self-descriptive messages.

In this iteration, the operation-to-resources gap has to be bridged. We claim that the semantics of a particular operation can be mapped to correlated archetypal semantics in most cases, e. g., *CRUD entity*, *calculate something* or *query set of things*. Based on this assumption, RESTify defines a couple of archetypal semantics and so called *mapping strategies* for these semantics.

A mapping strategy describes to which request sequence an operation can be transformed. Furthermore, it defines (non-formal) preconditions referring to syntactic or semantic properties of an operation which need to be fulfilled to make the mapping strategy applicable.

In the simple cases the sequence consists of one request but sometimes multiple consecutive requests are needed where a prior request contains the identifier that the consecutive request is sent to. An example for such a request sequence for the archetypal semantic "create" is the creation of an entity with the GET-PUT pattern [18].

Iteration II defines some mapping strategies for common

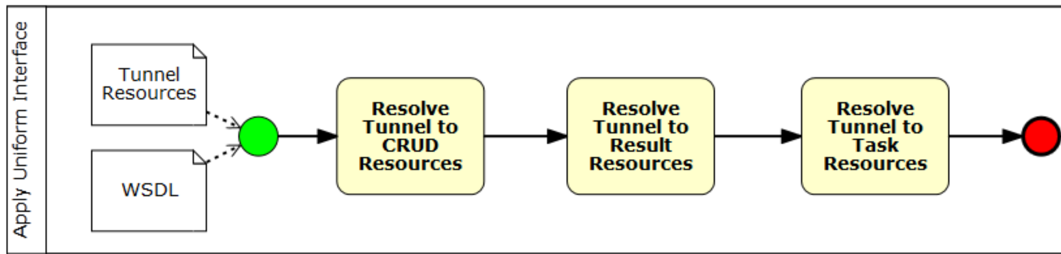

Figure 7: BPMN diagram of iteration II

archetypal semantics - starting with the most common case for web interfaces.

3.2.1 Resolve Tunnel to CRUD Resources

Analyzing nine randomly selected WSDLs leads to the result, that about 65% of the SOAP methods are CRUD methods of domain objects. The ratio per service differs from about 55% to 100%. Hence, starting with the resolution of tunnel resources into entity resources seems obvious.

The activity encoded explicitly in every resource of iteration I (or implicit in the corresponding operation name) is an indicator to decide, if the semantics can be simplified to CRUD. But in the end, the designers knowledge about the service semantics are decisive.

An operation, which may be an archetypal CRUD is restricted to be *strictly CRUD*. This means, that one entity of an entity set is being created, read, written or deleted by its identifier only. Especially arbitrary read semantics are specifically excluded and will be handled by other tasks.

Once an archetypal CRUD semantic is identified for any operation, a mapping scheme has to be selected for every identified CRUD operation (or its correlated tunnel resource). As an example, possible mapping schemes (respectively request schemes) for CREATE are:

- `POST /{set}`
- `PUT /{set}/{identity}`
- `GET /{factory}/{token}` → EntityURI
  `PUT` EntityURI

Applying a scheme to a specific operation means, that at least one variable of the URI Template hat to be instantiated. The resulting URI Template may degenerate to an URI, describing a singleton resource of the system. If the resulting URI Template contains one or more variables, it describes a set of resources.

The operations *CreateHIT, DisposeHIT, GetHIT* (see Figure 3) are good candidates for strict CRUD resources.

3.2.2 Resolve Tunnel to Result Resources

Another set of operations, which are not covered by the previous "strictly CRUD" mapping strategy, could be handled in this task. If the operation processes a simple function on objects of a domain, it can be mapped to "result resources". This covers a wide range of semantics, e. g., calculating a route between two places, filtering a known set of entities, searching for information.

One option is to apply a GET-strategy on result resources, which are defined by the operation. An operation has to be "safe" to apply this strategy. A flexible URI-Template can be expressed through the expansion modifier [7]. With the scheme `/{result}{?context*}` different URI-Templates can be covered:

- `GET /route?from={from}&to={to}`
- `GET /hits?type={hit-type}`
- `GET /search-results?term={term}`

A naming for the expected result resource set can be syntactically extracted from the WSDL, e. g., from the operation name or the result type, or from the documentation.

For large inputs, this mapping could be inapplicable due to practical limitations. URIs could become very long, which is restricted by some HTTP implementations. Allamaraju supposes using POST for this situations (see recipes 8.3 and 8.4 [2]). This can also be defined as a mapping strategy with an example URI like `/{processor}`. The context has to be moved to the request's HTTP body.

For the example excerpt, these strategies can be applied to the *SearchHits* operation or its tunnel pendant `/hits?action=search`.

3.2.3 Resolve Tunnel to Task Resources

Some operations indicate a "process-critical" state change of a domain entity. This is the case for the *Hit* tunnel resource set with the remaining, unmapped actions (*disable*, *extend*, *force-expire*), and *set-as-reviewing*).

An option is to resolve the tunnel into a task resource. The URI design proposed by the prototype is inspired by the jOpera [13] project:

- Task resources: `/task/{name}`
- Task instance resources: `/task/{name}/{instance}`

To indicate the different meaning, the naming should be also transformed, e. g., from `/hit?action=extend` to `/task/hit-extension/`. The prototype uses a simplistic algorithm to propose such conversions from verbs to nouns. The algorithm uses prioritized lists of typical verb-endings and conversions, e. g., "...end" to "...ension" or "...ize" to "...ization".

3.2.4 Results

After this iteration the different resources support different operations and are no longer URI tunnels but map to entities (see Figure 8). By applying the different strategies to the WSDL excerpt of Figure 3, the resource design documented in Table 1 can be developed.

Resource Description	Resource Identifier	Supported Operations	Corresponding WSDL Operation
HITs entity set	/hits/	POST	CreateHIT
filtered HITs result set	/hits?title={title}&reward={reward}&...	GET	SearchHITs
HIT entity	/hit/{hit-id}	GET DELETE	GetHIT DisposeHIT
disable HIT task	/task/hit-disabling	POST	DisableHIT
disable HIT task instance	/task/hit-disabling/{instance-id}	GET PUT	
extend HIT task	/task/hit-extension	POST	ExtendHIT
extend HIT task instance	/task/hit-extension/{instance-id}	GET PUT	
expire HIT task	/task/hit-expiration	POST	ForceExpireHIT
expire HIT task instance	/task/hit-expiration/{instance-id}	GET PUT	
review HIT task	/task/hit-reviewing	POST	SetHITAsReviewing
review HIT task instance	/task/hit-reviewing/{instance-id}	GET PUT	

Table 1: Extract of the resource design after iteration II

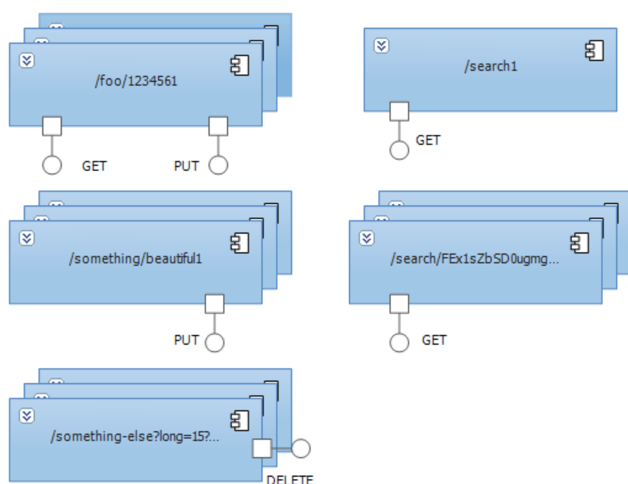

Figure 8: Example of resources after iteration II

The proposed tasks in this iteration do not claim any completeness to map every possible operation type, but task resources can be used as fall-back strategy for most operations. This means, there could be a better option to map a specific archetype semantic to support a versatile resource design with better caching or hypermedia support. Thus, RESTify suggests a formalized way to describe mapping strategies in an extensible and general manner.

3.3 Iteration III: Enable Hypermedia

To follow hypermedia as the engine of application state (HATEOAS), the last iteration (see Figure 9) introduces additional semantics to the RESTful design in comparison to the RPC-oriented design. By defining state-depended relationships among entity types and activities the interface description can be enriched with hypermedia.

3.3.1 Define Resource States

To complete the resource design this step defines "process-critical" states. Changing a resource's property ends always in a new resource state regardless of the property's importance. The states, which should be defined in this task, are states which may change available options in the domain's processes. Furthermore, entity states can be orthogonal to each other. Thus, an entity of a certain domain may have multiple states, while from a technical perspective a resource has only one valid state.

The simplest approach is to define only the states itself. A better option would be state machines, which describe the life cycle [15]. For the HIT example, some states are already defined implicitly by the task resources, which change a HIT resource to a certain state. These states are `extended`, `expired`, `disabled`, and `reviewed`.

According to the state definition, additional resources can be also introduced here. For example:

- `/disabled-hits`
- `/extended-hits`
- `/expired-hits`
- `/reviewed-hits`

With explicitly defined states at hand, resource relationships can be defined more appropriately.

3.3.2 Define Entity Resource Relationships

The first task of this iteration is to define typical relationships in the resource design of iteration II. Some relationships are static, thus valid in every state of a resource, and some are dynamic. The classification of Tilkov[16] can be used to define the static relationships among the classified resources (e. g., primary-, sub-, and list-resources). Every relationship between specific resources can be represented by a hyperlink, which follows a meaningful link relation type.

The resource design at this point describes resource sets of the domain by using URI Templates. Thus, it is reasonable to describe the relationships among the resource sets by link relation types between these templates.

The definition of a relationship can be defined by a simple triplet: `(source[state], link type, target[state])`.

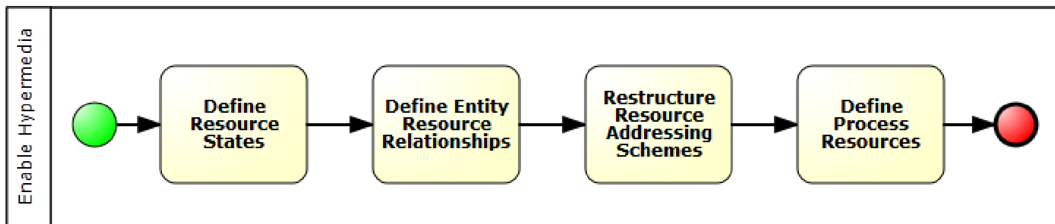

Figure 9: BPMN diagram of iteration III

`Source` and `target` are represented by URI-Templates, while `[state]` describes an optional requirement, that the resource has to be (or must not be) in one or more specific states to apply the hypermedia link. Multiple states are comma-separated and negation is defined by the symbol ¬.

This excerpt of HIT relationships uses link relation types from the IANA[7] registration and custom defined types[8].

- `("/hits", "item", "/hit/{hit-id})"`
- `("/disabled-hits", "item", "/hit/{hit-id})" [disabled])`
- `("/extended-hits", "item", "/hit/{hit-id}" [extended])`
- `("/expired-hits", "item", "/hit/{hit-id} [expired])"`
- `("/reviewed-hits", "item", "/hit/{hit-id} [reviewed])"`
- `("/hit/{hit-id}", "collection", "/hits)"`
- ...

Based on the results of iteration II, only few proposals can be made by the prototype here. For example, the CRUD resource design implies entities and related entity sets. This knowledge can be used to propose default relationships. By completing this task, the next step "refactors" the resource design.

3.3.3 *Restructure Resource Addressing Schemes*

Based on the definitions of the preceding task, the URI schemes can be redefined to reflect the hierarchical relationship. For example, the `/hit/{hit-id}` resource set and the list resource `/hits/` are hierarchical related. Therefore, the defined URI templates for *hit* entities can be subordinated to `/hits/`. Even though this is a cosmetic step, it could add coherence to the URI design. The prototype can support this process by analyzing the relationships and inferring a simple, hierarchical URI structure from the defined URI Templates.

3.3.4 *Define Process Resources*

If activity resources are defined, they can be set into a context of a process. These processes can be known implicitly by the designer or explicit mentioned in the SOAP service's documentation. The modeling of processes based on the activity resources can be done with approaches like jOpera [13].

[7] `http://www.iana.org/assignments/link-relations/link-relations.xml`

[8] CURIE syntax, ex := `http://www.example.com`

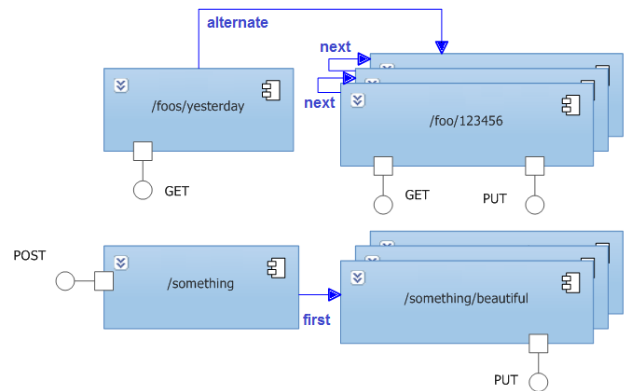

Figure 10: Example of resources after iteration III

3.3.5 *Define Activity Resource Relationships*

As a last step, the entity resource sets are linked with possible activities in their context. Usually, such activities are in general state-dependent. A representation of a *paid* order resource should not contain a link to a payment resource.

For the HIT example, the following activity relationships can be defined:

- `("/hit/{hit-id}" [¬disabled,¬expired], "ex:disabling", "/task/hit-disabling)")`
- `("/hit/{hit-id}", "ex:reviewing", "/task/hit-reviewing)"`
- `("/hit/{hit-id}" [¬expired], "ex:expiration", "/task/hit-expiration)"`
- `("/hit/{hit-id}" [¬disabled], "ex:extending", "/task/hit-extending)"`

3.3.6 *Results*

Iteration III produces just a few new resources, according to defined states of interest. The main goal of this iteration is, to enrich the resource design with hypermedia (see Figure 10), and thus to forward a loosely-coupled design as well as HATEOAS.

4. CONCLUSIONS AND FUTURE WORK

We presented a procedure model to transform a SOAP design to a RESTful HTTP design. Preconditions for the procedures are an interpretable service description (WSDL) and the implicit knowledge of the service operations' semantics. The procedure is partly implemented in a prototypical

web application where the presented examples are extracted from. Some tasks can be supported by simple algorithms.

The state of iteration I is very stable and produces a reasonable design. Iteration II reflects best the gap between RPCs and resource-oriented design. Although this iteration will define more sophisticated tasks in the future, the basic idea of *mapping strategies* will remain. More differentiated strategies will be defined, if more SOAP services are evaluated. The last iteration however, is ongoing work and the proposed procedure model is still in an experimental state. Furthermore, there are important aspects, which are not fully mentioned here, e. g., usage and design of media types, caching definitions, or authorization scenarios.

But looking at the first results produced with the prototypical application, we expect an iterative procedure model to be a successful approach to close the design gap between the RPC- and REST-style. Furthermore, the procedure model can be possibly applied with minor changes to object-oriented interfaces. We do not expect nor propose this model to create a fully functional and reasonable implementation based on its results. There are many aspects on how to deal with the resource granularity, e. g., performance and caching issues. But we think an iterative approach can enhance the learning process of REST aspects or respectively the "unlearning" of RPC concepts, and that the results can be used as a debatable design basis.

The result of every iteration can be used as a basis for a manual service or wrapper implementation, but at least the documentation can be generated out of the iteration results.

5. REFERENCES

[1] J. Algermissen. Classification of HTTP-based APIs. http://nordsc.com/ext/classification_of_http_based_apis.html, Feb. 2010. [accessed 2012-02-14].

[2] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly Media, 2010.

[3] S. Allamaraju. Measuring REST. http://www.subbu.org/blog/2011/05/measuring-rest, May 2011. [accessed 2012-02-14].

[4] M. Athanasopoulos and K. Kontogiannis. Identification of REST-like Resources from Legacy Service Descriptions. *17th Working Conference on Reverse Engineering*, pages 215–219, 2010.

[5] T. Erl. *SOA design patterns*. Prentice Hall, 2009.

[6] M. Fowler. Richardson Maturity Model. http://martinfowler.com/articles/richardsonMaturityModel.html, Mar. 2010. [accessed 2012-02-14].

[7] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI Template. Internet-Draft. http://tools.ietf.org/html/draft-gregorio-uritemplate-08, Jan. 2012. [accessed 2012-02-14].

[8] M. Kelly. HAL - Hypertext Application Language. http://stateless.co/hal_specification.html, Oct. 2011. [accessed 2012-02-14].

[9] S. Kennedy, R. Stewart, P. Jacob, and O. Molloy. StoRHm: a protocol adapter for mapping SOAP based Web Services to RESTful HTTP format. *Electronic Commerce Research*, 11:245–269, 2011.

[10] M. Laitkorpi, P. Selonen, and T. Systa. Towards a Model-Driven Process for Designing ReSTful Web Services. In *ICWS '09: Proc. Int. Conf. on Web Services*, pages 173–180. IEEE, 2009.

[11] O. Liskin, L. Singer, and K. Schneider. Teaching old services new tricks: adding HATEOAS support as an afterthought. In *Proceedings of the Second International Workshop on RESTful Design*, pages 3–10. ACM, 2011.

[12] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu. Reengineering Legacy Systems with RESTful Web Service. In *COMPSAC '08: Proc. Int. Software and Applications Conf.*, pages 785–790. IEEE, 2008.

[13] C. Pautasso. Composing RESTful Services with JOpera. In *Software Composition*, volume 5634 of *LNCS*, pages 142–159. Springer, 2009.

[14] M. Porter. The Porter Stemming Algorithm. http://tartarus.org/~martin/PorterStemmer/index.html. [accessed 2012-02-14].

[15] S. Schreier. Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design*, pages 15–21. ACM, 2011.

[16] S. Tilkov. *REST und HTTP: Einsatz der Architektur des Webs für Integrationsszenarien*. dpunkt.verlag, 2009.

[17] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[18] E. Wilde. Creating Resources with GET/PUT. http://dret.typepad.com/dretblog/2011/11/creating-resources-with-get-put.html, Nov. 2011. [accessed 2012-02-14].