Case Study: Extracting a Resource Model from an Object-Oriented Legacy Application

Christoph Szymański
SAP AG
69190 Walldorf, Germany
christoph.szymanski@sap.com

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

ABSTRACT

Many companies have invested in legacy applications and want to benefit from the interoperability that the architectural style Representational State Transfer (REST) offers without redeveloping their software. One of the crucial parts when adding a REST interface to an existing application is creating an appropriate resource model. Utilizing any available model of the legacy application can accelerate development significantly because existing domain knowledge, data, and business process implementations can be reused. Despite the maturity of the architectural style, there is still little record of creating a resource model from existing object-oriented applications. This article presents a lightweight modeling process: First we harvest an existing object model for resource candidates, afterwards the resulting model is enhanced incrementally until a suitable resource model emerges. The process is illustrated by a case study that highlights interesting challenges, such as a comprehensive domain model and long running processes, as well as pragmatic solutions for these challenges. The paper demonstrates that it is feasible to add a RESTful interface to a legacy application even in a process rich environment.

1. INTRODUCTION

In many companies there is an urge to expose the data and the processes of legacy applications as web services. In the last years RESTful web services have become a common choice because of their inherent interoperability, scalability, and the use of the Hypertext Transfer Protocol (HTTP) [8]. Accordingly, the need to efficiently produce dependable resource models to build good RESTful APIs becomes increasingly imperative.

There is work on resource modeling directly from a problem domain [6, 7], but many legacy applications already have well defined and documented models. Applications programmed in the object-oriented paradigm have an object model that provides a solid basis for a resource model. While at a first glance the transformation of an object model to a resource model seems trivial, because both deal with entities and message exchange, the paradigms differ in some key aspects. The uniform interface of a resource model probably being the most significant: In HTTP there is only a limited and standardized set of methods with fixed semantics available and also the messages themselves are defined, e. g., using media types and standardized return codes. Another difference lies in the relationships between entities. Object relationships are usually carefully analyzed to avoid unnecessary coupling. In contrast, the Hypermedia as the Engine of Application State (HATEOAS) principle [4] in RESTful designs depends on rich linking between resources to facilitate easy service traversal and good connectivity between different services. As a result the focus during the modeling phase moves from defining object method signatures and relationships to designing the resource state and the messages used for communication between client and server.

The legacy application in this case study—the *Virtual Infrastructure Management (VIM) Portal*—is an internal web application at SAP. It has been developed to administrate and monitor a large number of virtual machines (VMs) used in various stages of the software engineering process. These VMs are used for testing, debugging, hosting build tools, and as training systems. VMs are suitable for these scenarios because they can be easily reset, deployed quickly, and teams in different timezones can utilize the underlying hardware effectively. The VIM Portal is programmed in PHP using the language's object-oriented features[1]. The subject of the case study is to document the creation of a resource model for the VIM Portal using its object model. We use this resource model for implementing a RESTful API, which replaces a set of non-standard legacy web services. The goal of the new API is to expose existing VIM Portal features primarily for future automated clients.

After introducing related research in the next section, we introduce the case study and motivation for a RESTful API in Section 3. The suggested transformation from object- to resource-oriented models is described in Section 4 and illustrated with examples from the case study. We conclude this article with our findings and challenges for future work.

2. RELATED WORK

Different approaches and case studies exist about transforming legacy applications to RESTful APIs. Engelke and Fitzgerald [3] describe why and how an existing legacy application for Internet bidding has been generalized and replaced by a RESTful API. They focus on authentication and

[1] `http://www.php.net/manual/en/language.oop5.php`

WS-REST 2012, April 2012; Lyon, France

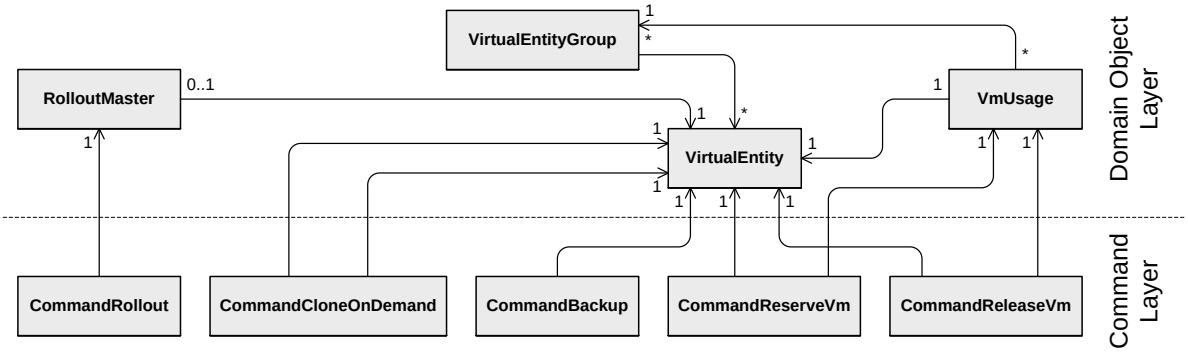


Figure 1: Detail of the object-oriented source model showing several commands and their relationships with domain objects.

security issues and the service offers only CRUD operations. Instead of reusing existing implementations they designed a new protocol and implemented it in different languages. The case study of Fuentes-Lorenzo [5] provides insights on how existing data can be managed using RESTful web services. Again the focus is on data-centric CRUD operations.

Liu et al. [7] describe a common process for re-engineering legacy systems to RESTful services which is focused on data- or entity-driven systems. They identify resource granularity as the key problem and focus very much on the Uniform Resource Identifier (URI) design. Their process is not bound to any particular source model. Parts of their process have been used to derive the process presented in this work, but handling of complex and long running processes had to be added. Some work has been done on modeling resource-oriented applications: Laitkorpi et al. [6] present a model-driven process for RESTful Services starting with a functional specification and not an existing system. They focus on sequence diagrams and the communication between server and client to extract the resources instead of analyzing object models.

Tilkov [10] provides ideas for resource classification used in this work. Some aspects from Tilkov's book are also discussed in [9] where the classification is being incorporated in a more complete metamodel for resource-oriented models.

3. THE CASE STUDY

After introducing the VIM Portal and some of its use cases, we outline the requirements and reason why a RESTful HTTP based implementation would meet the demands.

3.1 The VIM Portal

The VIM Portal provides a web interface to access VMs hosted on a distributed virtual infrastructure (VI) provided by third party vendors. To share a more detailed view we describe two typical use cases of the application briefly.

The first is the most common interaction of a user with the VI: Using a VM includes locating a suitable VM in the system, starting it on the VI, and registering it as being used to ensure exclusive use for this user. After finishing work on the VM it is released, i. e., stopped and the usage is canceled. For later reference we call this use case UC1.

The second use case is an administrative task (UC2). A configured *master VM* can be replicated to several clones. Such a *roll out* is used by designated service teams which install the latest version of a software on such a master VM which is then being rolled out to a number of VMs that can be utilized by other team members.

Figure 1 shows parts of the legacy application's object model with focus on the details needed during modeling. It is arranged around the class `VirtualEntity`, which represents a VM, some adjacent domain classes, and commands. `CommandRollout` implements the task of mass cloning a VM acting as `RolloutMaster` (UC2). The normal usage data of a VM is available in `VmUsage`. Reserving and releasing a VM (see UC1) is modeled in two commands, `CommandReserveVm` and `CommandReleaseVm`. VMs can be organized in `VirtualEntityGroups`, which are used to configure usage restrictions.

These classes are embedded in the domain object layer and the command layer. The VIM Portal's layers are described in more detail in Figure 2. Domain Objects provide operations to access entity data in the persistence layer as well as entity specific domain logic. Domain objects are used in the adjacent view and service layers. Services encapsulate meaningful processes that can be combined and used in complex commands. Commands act as the request target for clients to execute an action. Commands control the execution by checking prerequisite and creating domain objects used in the process. Furthermore, they can set up process logging and use the application's Access Control List (ACL) to authorize the client.

3.2 Challenges

Several challenges had to be considered designing the new web service. One goal was to reuse existing code and business logic which has been running reliably for years. Another early decision was to reuse infrastructural features like the ACLs for authorization and HTTPS for authentication.

After collecting the requirements from existing and potential users of the service we decided to use a media-type agnostic architecture: Whereas most clients are able to produce and consume any flavor of XML for communication, there are web clients—including the VIM Portal's web UI itself—that prefer to consume JSON for AJAX calls. Extendability to serve any format in the future had to be taken into account.

Furthermore, migration of existing clients and the creation of additional clients in different programming languages and environments has to be made as easy as possible. A lightweight architecture and protocol with library support in the most common programming languages is necessary.

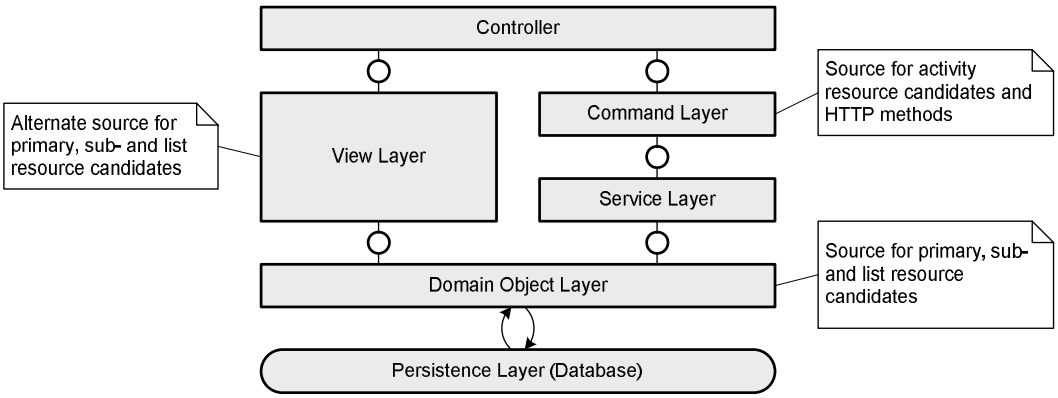


Figure 2: VIM Portal application layers as the source of resource candidates.

The new web service would have to support long running tasks as well, which is one of the most important requirements in a business domain heavily depending on slow external resources.

3.3 RESTful HTTP

The REST architectural style fits the requirements, most notably the benefits deducible directly from Fielding's architectural constraints [4]: *Scalability* is required, namely by caching and distributing the service across multiple servers because the application manages more than 4,000 virtual machines for corporate wide access and is still growing. Data from slow external source like the VI must be *cached* to reduce requests and avoid performance problems. Consequently, all server responses should make use of cache control header information to give clients a hint how long the data is valid.

New features need to be added to the VIM Portal regularly, requiring that the web service supports *modifiability* in several ways. The *uniform interface*, i. e., the elements of HTTP, self documenting resources and hypermedia as the engine of application state will simplify the service's consumption, thus encouraging the creation of clients and the reuse of data and processes.

4. MODELING PROCESS

First we introduce and motivate a few assumptions prior to modeling. Afterwards the modeling process is described. Examples and experiences from the case study accompany the modeling steps.

4.1 Early Modeling Considerations

We assume that it is beneficial for implementation and modeling to classify the resources. Using resource types makes implementation and documentation of the resources easier by revealing their structural function. Most of the resource types introduced in Tilkov [10] are used during modeling: *Primary resources* are entities in the business domain, similar to domain objects in classic software design. *Subresources* are part of a primary resource. *List resources* are used for listing and creating new resources. They can be *paged* and *filtered* because of potentially large result sets. *Activity resources* model business processes and tasks.

The VIM Portal's layers are described in Figure 2. The command layer is the source for activity resource candidates, because one commands encapsulates one action in all user work flows. User authorization to execute commands is managed in the application's ACL. The services on the other hand implement single steps which can be reused and combined in complex commands. Services can not be consumed by a client directly. Primary and subresource candidates are extracted from the domain object layer. An alternate source for resources is the view layer because it reveals what data needs to be published. The domain layer has been chosen instead because the extraction can be automated more easily.

4.1.1 Primary Resources

Central domain entities of the application are classified as primary resources. They provide the most basic and frequently accessed entity data.

All primary resources are supplemented with a list resource. We use the common practice to create resources by sending a `POST` message to the corresponding list [1, 10].

4.1.2 Subresources

Subresources are modeled depending on various factors: One indicator is the object relationship in the source model. If the relationship of two objects is a composition, then one is modeled as a subresource of the other. In Liu et al. [7] there is an interesting approach to generate URIs from relationship types. In contrast our approach is less sophisticated and not directly aimed at generating URIs but to indicate resource types.

Another factor to consider is that data semantically belonging to one resource is sometimes better distributed over separate resources if the data requires different cache settings. This can be the case if a resource consists of data from different sources, e. g., from a slow external source like the VI in this case study and a fast database.

4.1.3 Activity Resources

Many process-rich web applications feature a service layer implementing business processes. It is expected to be one of the major challenges to map those processes canonically to the concrete resource candidates. For example, there are several processes creating new VMs, but only one of them can be mapped to the `POST` operation of the corresponding list resource. Besides, these processes differ too much semantically to just distinguish them by the request repre-

sentation.

We decide to handle long running processes asynchronously allowing the client to get an immediate response. We create separate activity resources for this purpose: The creation of an activity provides the client with the location of the new activity which can be used to monitor the progress of the tasks.

4.2 Modeling Steps

Liu et al. [7] address key problems that arise during the creation of RESTful resource models from legacy applications: Identifying resources, mapping services to the HTTP verbs, URI design considerations and API description. We take elements from their approach [7] and metamodel elements [9] and combine these into a simple modeling strategy. The modeling process consists of four phases: Identification and extraction of resource candidates, refining the model, consolidation of activities and a final step of validation including the design of an entry resource. In contrast to Liu et al. [7], we treat URI design as a secondary task because the URI can be deducted from the resulting resource model easily.

4.2.1 Extraction of Resource Candidates

In a simple examination the resource candidates are extracted from the source model. Domain classes are considered for primary resources, subresources and lists, classes from the command layer for activities. Each resource candidate's relevance for the resource model is discussed with stakeholders with regard to existing and future clients of the new API. As a result, all classes in Figure 1 are used as resource candidates for the resource model. Only very few domain classes are omitted, most of them representing technical details which are used in the application's internal processes without user interaction.

In this case study 27 activity resource candidates and most of the source model's domain classes have been carried over to the preliminary resource model. All primary resources received an additional list resource. During the extraction from the source model the object relationships were preserved to serve as links in the resource model.

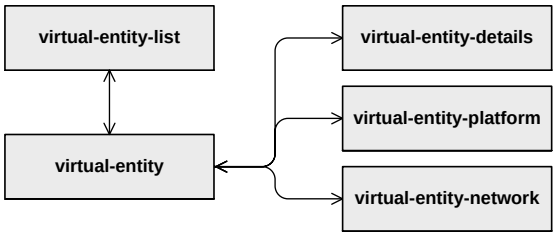
4.2.2 Refining the Model

During this step the partitioning of the resources is reviewed. Some resources need to be split, some containing only little information need to be integrated. After these changes the names of the resource candidates need to be changed. Lowercase resource names instead of *CamelCase* emphasize the transition to the resource paradigm and are considered more URI-friendly.

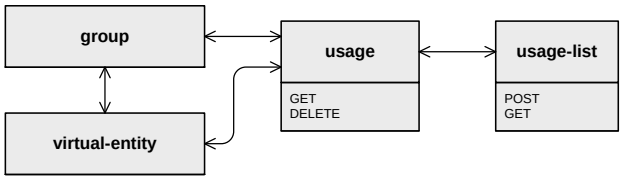
Figure 3a shows that the information regarding a `virtual-entity` resource is split across various subresources: Whereas most basic data remains with the primary resource, details and platform information have their own resource because they are only required in special use cases. The network information is also split off the primary resource because collecting this information involves slow external dependencies like the VI and company network services. This avoids fetching the data when it is not required by the client. Additionally, using different cache control headers on *expensive* resources to indicate longer cache entry life time can improve client performance and save server resources.

Relationships between resources are not treated as restrictively as in object-oriented models. Short navigational paths for clients are preferred. All links between resources in this case study are by default bidirectional to provide many options to traverse the resource graph quickly.

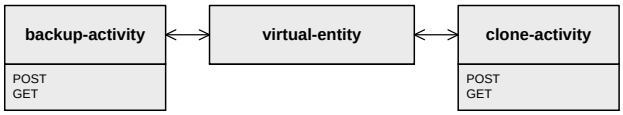
There is a possible feedback loop from this phase to the object-oriented source model: If a lot of renaming or splitting and merging of resources is necessary and the changes are not merely indicated by REST requirements, these changes should be applied to the source model as well.



(a) Model of `virtual-entity` and its list- and subresources.



(b) The processes from UC1 are mapped to `POST /usage` and `DELETE /usage/{id}`.



(c) Two of the remaining activities.

Figure 3: Resources and links from the VIM Portal model.

4.2.3 Consolidation of Activities

In this phase we try to map as many activities as possible to a resource's request methods. Only the remaining activities are modeled as activity resources.

Many of the VIM Portal's processes are simple CRUD operations on a database and can be mapped to the HTTP verbs easily. The remaining processes interact with various subsystems of the VI and are therefore potentially long-running. From the 27 activity resource candidates all but three could be attached to an appropriate request method of a concrete resource, e. g., `CommandReserveVm` and `CommandReleaseVm` in Figure 1. These two legacy command classes implement parts of UC1 (see Subsection 3.1), and both can be mapped to `POST /usage/` and `DELETE /usage/{id}` respectively as shown in Figure 3b.

The three remaining resources are long running cloning tasks, e. g., the commands `CommandCloneOnDemand` and `CommandBackup` from the source model (see Figure 1) and are represented as activity resources in Figure 3c. The class `CommandRollout` in Figure 1 is part of UC2 (see Subsection 3.1). Since this process needs asynchronous processing,

Relation Name	Usage
`self`	Refers to the current document.
`up`	Used in subresources to link to its primary resource.
`next`, `previous`	Paging list resources.
`help`	Providing human readable resource help.

Table 1: Examples of IANA link relations reused in the VIM Portal.

it is modeled as an activity resource as well.

These results indicate that few HTTP verbs suffice to create a process rich and expressive interface.

4.2.4 Finalizing the Model

Similar to a service description in traditional web service architectures we model an entry resource. This resource provides useful links to different areas of the service.

Finally, we verify that all modeled resources are connected and reachable from the entry resource. Early feedback from stakeholders indicated that some paths between resources were too long. To rectify this, additional links between resources are introduced where necessary.

4.2.5 URI Design

As we expect mostly automated clients, we set out for simple, yet comprehensible URIs. Readable URIs are important to help developers and other human users understand basic resource and relationship semantics when exploring the service.

The lowercase, hyphen-separated resource names from the model are also used in the URIs. To identify a resource the unique identifier is appended to the resource name (`/{resource}/{id}`), reusing the database keys of the base application, which guarantees their uniqueness. This URI design is used for primary and activity resources. Subresource URIs have been designed to emphasize the relation to their primary resource, e. g., for the subresource `virtual-entity-network` from Figure 3a: `/virtual-entity/{id}/network`

Lists are represented by the URI of their respective primary resource without the trailing identifier string. To enable filtering and paging the URI query component [2] is utilized:

```
/{list}?page={number}&{filter-key}={filter-value}
```

4.2.6 Link Relations and Media Types

Resources are linked using standard IANA link relations[2] where possible, e. g. `up` for linking primary resources from their subresources, or `help`. Table 1 shows the IANA link relations reused in the VIM Portal API. Most link relations, however, are service specific, therefore there is not much room to reuse the generic standard link relations.

The design of media types is regarded as a key issue in resource modeling [6]. An important requirement for this case study is to keep the implementation as simple as possible. Therefore, media types must be easily producible and consumable by the service. This is achieved by assuring that the media types are structurally equal.

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

The VIM Portal's API is able to produce and consume custom XML representations for each resource. Additionally, `application/x-www-form-urlencoded` is accepted as a request only media type. These media types are documented and the documentation is available for client developers.

5. FINDINGS AND FUTURE TASKS

With the described process the resources, their linking, the contained information, and URIs can be extracted from a source model and adapted if necessary. The benefit of this approach is the reuse of existing knowledge combined with the flexibility to adjust the source model to the requirements of REST. This approach can be used for other applications as well since the VIM Portal's key characteristics—the object model, a service layer implementing business processes, dependencies to externally source that are potentially slow and expensive to call—also apply to other enterprise web applications.

The resource type classification proved to be particularly useful. The concept of activity resources facilitates the modeling of processes as resources in cases where long running tasks require asynchronous processing. Additionally, it helps to derive the URIs. This can counter a common objection against RESTful HTTP stating that the set of standard methods is too limited and not expressive enough to be suitable in complex problem domains. During modeling the number of activity resources could be reduced to only three—as a result the full range of the resources' HTTP verbs is used.

A main obstacle is the lack of a modeling language like UML for REST models. Object models are very different from resource models. UML diagrams are not able to convey the many different details of a resource model where information like URI, caching directives, content types, or filters for lists are important, whereas method signatures are standardized.

Another interesting finding is that the initial harvesting of the source model is a very mechanic, rule based process which can be automated. Implementation showed that after defining the resource model implementing the API was very mechanic as well. This leads to the assumption that the actual program code could be generated almost completely from an expressive model into a REST framework. With better tool support, developers could focus more on modeling the problem domain without having to bother about REST subtleties, which would consequently result in higher quality and more compliant RESTful API.

6. REFERENCES

[1] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly Series. O'Reilly, 2010.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Technical Report 1738, Internet Engineering Task Force, December 1994.

[3] C. Engelke and C. Fitzgerald. Replacing Legacy Web Services with RESTful Services. In *WS-REST '10: Proceedings of the First International Workshop on RESTful Design*, 2010.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, 2000.

[5] D. Fuentes-Lorenzo. Managing Legacy Telco Data Using RESTful Web Services. In E. Wilde and C. Pautasso, editors, *REST: From Research to Practice*, pages 7 – 26. Springer, 2011.

[6] M. Laitkorpi, P. Selonen, and T. Systä. Towards a Model-Driven Process for Designing ReSTful Web Services. In *IEEE International Conference on Web Services*, pages 173 –180, 2009.

[7] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu. Reengineering Legacy Systems with RESTful Web Service. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 785 –790, 2008.

[8] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Series. O'Reilly, 2007.

[9] S. Schreier. Modeling RESTful Applications. In *WS-REST '11: Proceedings of the Second International Workshop on RESTful Design*, 2011.

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