Resourcebus
A New Substrate for Model-driven Creations

Petr C. Smolik and Pavel Vitkovsky
Metada s.r.o., Antala Staska 64, Prague, Czech Republic

Keywords: Model-Driven Engineering, Resource-oriented Computing, Model Versioning, Model Storage, Model Interpretation, Linked Data Platform.

Abstract: In this paper we present our on-going work on a framework we are building as a basis for construction of modeling and metamodeling environments. Resourcebus is basically a web-based resource-oriented computing environment for creation and execution of dynamic resources based on interpretation of models. Resourcebus adheres to the linked data principles and REST architectural style, and enables publishing models on the web using open standards. The framework includes built-in versioned storage, caching, access control, and set of various interpreters. We are developing Resourcebus primarily as a runtime environment for model storage, model interpretation, and code generation. Resourcebus itself knows nothing about metamodels and does not implement any particular meta-metamodel. It just provides an environment for creating them. We conclude this paper with a list of issues that still need to be resolved.

1 INTRODUCTION

There are various types of model-driven tools available. Some tools, such as Xtext (Eysholdt, 2010), are primarily text-based and enable development of textual DSLs for various purposes and include code editors with syntax coloring, and code completion. Some tools, such as MetaEdit+ (Tolvanen, 2006), are primarily graphical and enable creation of visual representations of domain-specific concepts and their relations. Finally, some tools, such as the Meta Programming System (MPS) (Voelter, 2011), enable textual projectional editing via structured code editors that directly operate on the syntax tree of a program instead of on lines of text. A large recent comparison of various tools and approaches is provided in a report from the 2013 Language Workbench Challenge (Erdweg, 2013).

In the spectrum of approaches, Resourcebus is a foundation for web-based model-driven tools that are mostly targeted to non-programmers. Users of such tools create models primarily by filling forms, but there is also some support of graphical navigation through the models and there are structured code editors for specifics such as expressions.

Resourcebus realizes several foundational services that model-driven solution creators should not have to worry about. These are model storage, versioning, intelligent caching, access control, multi-user environment, support for combining languages (both domain-specific and general-purpose ones), facilities for model interpretation, and the possibility of publishing models and metamodels on the web.

In this paper we show our work in progress on Resourcebus. We are currently working on release 0.14 and hope to reach version 1.0 within months. Parts of Resourcebus are already in use by two of our customers in the financial services domain, since we are in the process of incremental migration of our existing Metada Metarepository metamodeling environment to Resourcebus. Metarepository takes advantage of all the Resourcebus features and adds its meta-metamodel and few model interpreters, such as an editor interpreter that enables form-based editing of models based on a specified metamodel.

2 FUNDAMENTALS

Resourcebus adheres to the linked data principles stated in (Berners-Lee, 2006):

- Use Uniform Resource Identifiers (URIs) as names for things.
- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using the standards such as RDF (Klyne, 2004).
- Include links to other URIs, so that they can discover more things.

Using these principles, Resourcebus enables expressing models and metamodels as regular linked data and enables standard ways of their creation, maintenance, and retrieval as regular web resources.

2.1 Resource

Resourcebus resource is defined as a *thing* identified by an HTTP URI. Using content negotiation some representation of the *thing*, or information about the *thing* can be retrieved. This way Resourcebus has no problem working with model entities that are often representing real-world things or abstract things that have only some metadata about them, but cannot be really retrieved themselves.

The distinction whether resource identifies a real-world thing or an abstract concept and not a file or a document can be made by inspecting the available resource properties describing the resource. We found this solution more efficient and practical compared to other proposed solutions such as the use of "303 URIs" or "hash URIs" that are discussed in (Sauermann, 2008).

2.2 Resource Properties

Each Resourcebus resource may have one or more resource properties. Resource properties are directly related to the concept of RDF predicates and their types to RDF properties. Each property type is also a *thing* identified by an HTTP URI, so that information about any property may be retrieved and used by Resourcebus applications.

Resource properties are either simple or complex. Simple property has a simple value whereas complex property defines a sub-resource that can also have simple or complex properties. Each complex property has an ID unique within the scope of the given resource and may be addressed by adding a fragment identifier with its ID behind the resource URI (e.g. http://example.org/resource#cp).

Within Resourcebus, resource properties are natively represented in simple XML format, where XML namespaces are used to declare the full HTTP URIs of all the resource properties. Via content negotiation, it is possible to retrieve also other representations of resource properties such as in text/turtle (Beckett, 2013) or application/rdf+xml Klyne, 2004) formats.

This is a simple example of the native XML representation of resource properties:

```
<rbs:Data xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:ex="http://example.org/”
  xmlns:rbs="http://resourcebus.org/ns/storage#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
  syntax-ns#">
  <dcterms:title>Example resource</dcterms:title>
  <rdf:type>ex:ExampleType</rdf:type>
</rbs:Data>
```

And this is the above example represented in text/turtle:

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ex: <http://example.org/>.
@prefix rbs: <http://resourcebus.org/ns/storage#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
  syntax-ns#>.
<> a ex:ExampleType;
  dcterms:title "Example resource".
```

The example shows very simple properties of an "Example resource" of type ex:ExampleType. Information about the example type then would be retrievable, in non-example scenario, from a URL like http://example.org/ExampleType.

We are convinced that this form of representation provides easy and standard way of sharing models while enabling clear composition of concepts from different ontologies or metamodels.

2.3 Dynamic Resources

Resourcebus resources may have content. The content is either static or dynamic, or does not exist at all. If there is no content, there are at least resource properties informing about the existing resource. Static resource is a resource that has some associated file that can be directly retrieved as its content. Dynamic resource provides its content (or even properties) via interpretation of its properties by an interpreter (depicted on Figure 2). During interpretation it is possible to call other resources and this mechanism thus provides means for complex computations and even whole model interpretations.

We expect that dynamic resources may also provide elegant means for constructing modern web applications based on the REST architectural style (Fielding, 2002), because they may provide useful properties for their execution and navigation to related resources, and thus define their REST APIs. Furthermore, they may use APIs of other such resources to achieve their composition into complex applications.

2.4 Content Negotiation

Since Resourcebus resources may contain both properties and an actual content, it has to be clear how to retrieve each of them. The HTTP protocol does not provide explicit means for separate handling of content and metadata. For this reason Resourcebus uses the following rule during the HTTP content negotiation:

- IF there is no content nor properties THEN return 404 status code
- ELSE IF properties may be represented in an acceptable content-type AND URL does not end with a filename extension THEN return properties
- ELSE IF content may be represented in acceptable content-type THEN return content
- ELSE return 406 status code.

In other words, if Resourcebus client wants to retrieve resource properties it puts to the Accept header only the content-types expected to be used for properties representation, such as text/turtle or application/rdf+xml. Nevertheless, if a client wants to request content with such specific content types, it has to use a URL with corresponding filename extension (i.e. .rdf or .ttl).

To enable the REST architectural style, the actual resource properties returned do contain a property holding a reference to an existing content with the proper filename extension included, so the client can just follow this advice and does not have to resort to undesirable URL construction. Also, if content is returned on the resource URL without the filename extension, the response headers contain the content-location header that provides the URL with the proper filename extension included.

3 ARCHITECTURE

The architecture strategy of Resourcebus is to have a core that is as small as possible (like a microkernel) and all other functionality to be implemented via pluggable “interpreters” as shown on Figure 1.

Resourcebus interpreters implement the methods (corresponding to standard HTTP methods) of an abstract interpreter provided by the Resourcebus core. Interpreters may use the Resourcebus client API to access other resources based on their needs. Interpreters are used through dynamic resources where a dynamic resource is configured (via resource properties) to use a specific interpreter for its interpretation as shown on Figure 2.

Resourcebus core uses a storage API for

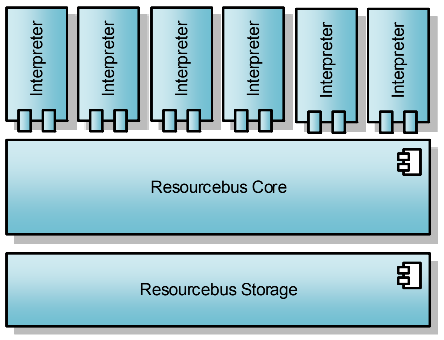

Figure 1: Resourcebus components.

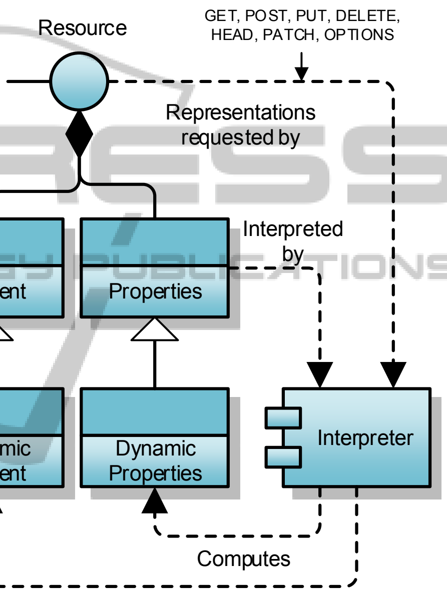

Figure 2: Resource-oriented computing.

persistence of both resource content and resource properties. The following subsections provide further information about the individual components.

3.1 Resourcebus Core

Resourcebus core is divided to client and server parts as shown on Figure 4. The client part is used to access both local and external resources via standard HTTP methods and respects the content negotiation rule specified in section 2.4 in order to distinguish resource content from resource properties. A resource-oriented API is provided that enables traversal of the graph of related resources instead of having to place raw HTTP requests.

The server part implements the resource handling via the standard HTTP methods. The server ensures communication with the storage where static resource content and resource properties are stored, executes dynamic resources by running the appropriate interpreters, and caches the resulting representations for repeated use.

The server tracks all resource execution dependencies and for each cached representation it knows exactly what was used for its computation. Representations have to be un-cached only if something changed that they depend on. Computations are thus done only when really needed and nothing is recomputed unnecessarily. This style of computing, depicted in Figure 2, based on our opinion, could be called the "resource-oriented computing", nevertheless this term is currently not widespread and one of the few used we know of is in (Geudens, 2012) where it is made more complex by using a non-standard protocol.

We see the resource-oriented computing style used within Resourcebus as its most interesting contribution for the area of model execution (both interpretation and code generation), because involved computations have to be recomputed only if their sources change. This means that not only model changes get effectively propagated, but even the metamodel or meta-metamodel changes do. This allows for construction of integrated metamodeling environments (aka metatools).

Currently there exists one metamodeling environment built on Resourcebus, the Metada Metarepository, that realizes a meta-metamodel and a set of metamodels and model interpreters realizing the metamodeling environment. The tool has been already used on two larger projects in the financial services domain.

The philosophy of Resourcebus core is also very similar to the one of the Linked Data Platform (LDP), as described in (Speicher, 2013), because Resourcebus is built on the same linked data principles. Resourcebus is mostly compliant with the current working draft of the LDP specification and it should not be a problem to reach full compliance to the potential standard in the future.

3.2 Resourcebus Storage

Resourcebus is designed to support large development teams working on relatively large models with tight and overlapping release-cycles. Versioning capabilities are therefore natively included in the storage API. Versioning of individual models or whole model repositories is possible. There is an instant access to models at any branch, tag, or historical commit. Private branches are used by individual users to model and debug models in a sandbox where they are not distracted by changes of other users. Private branches may be updated with changes from their parent branches and may publish their changes back to them. Propagation of fixes is possible from production branch to several development branches.

Currently there is just one implementation of the storage API, depicted on Figure 3, but other implementations are possible. It is based on the Git versioning system described in (Chacon, 2009) that we extended with the support of sparse checkouts. When a private branch is created, only a new Git branch pointer is created, when changes are made, only those changes are written to the sparse. No full checkouts are needed, which makes large number of private branches to be continually in use without multiplying the disk space needed. Apache Lucene (McCandless, 2010) indexing is used to support fast lookup of entities and traversing their relations. The storage format of resource properties is XML.

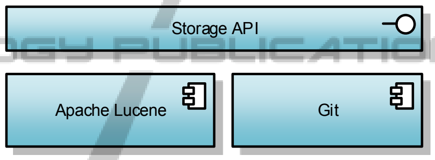

Figure 3: First storage implementation.

This storage implementation combining Git with Lucene was initially created only to figure out requirements for the storage API, but it seems to be still sufficient for a model repository with 30,000 objects (180MB in XML files), hundreds of tags and branches, and tens of thousands of commits.

More sophisticated storage implementations are possible if needed. Good candidates may be XML or NoSQL databases, both described in (Strauch, 2011). It is also possible that several different storage implementations will need to be used in parallel, since each one will have different trade-offs. For example the Git/Lucene implementation seems to be quite powerful on the versioning side.

The Apache Lucene search engine is useful also for full-text search both in static resource content or properties, but this feature does not have to be available through the storage API and could be implemented via separate interpreter.

3.3 Interpreters

Resource interpreters enable realization of dynamic resources. Their use is configured via the properties of a given dynamic resource. They are called interpreters, because their primary purpose is to interpret models and do useful things based on them. With different interpreters behind different resources it is easy to combine both interpreters and metamodels into flexible applications.

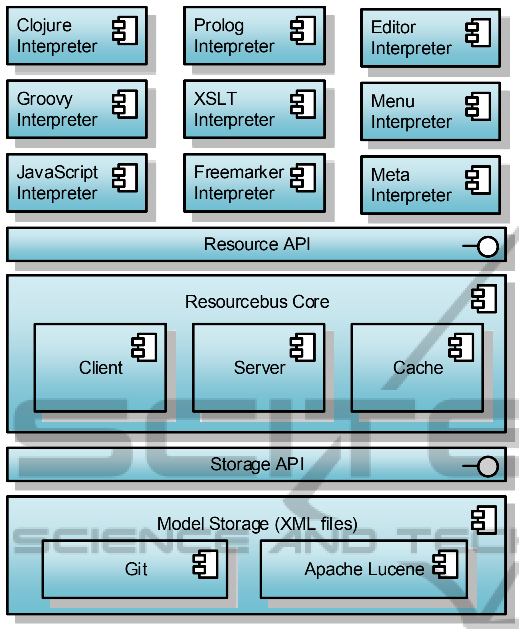

Figure 4: Resourcebus architecture.

Currently the most interesting interpreter realized is the Editor Interpreter used by the Metada Metarepository to provide for form-based editing of arbitrary models based on the interpretation of their metamodels. It nicely demonstrates how changes in metamodel immediately take effect in the runtime.

The same way that Resourcebus interpreters enable execution of various domain-specific languages, they may support execution of various general-purpose or special-purpose languages that can be this way easily used within Resourcebus and even use its client API to access other resources. We have used or at least tested interpreters for the following languages: Groovy (Koenig, 2007), JavaScript (Crockford, 2008), Scala (Odersky, 2010), Clojure (Emerick, 2012), Freemarker (Forsythe, 2013), XSLT (Kay, 2007), and XQuery (Boag, 2010).

We plan Resourcebus interpreters to be hot-pluggable and to enable parallel existence of their several versions. The former should allow for flexible dynamic configuration of applications, the latter should enable long-term support of older functionalities without having to migrate to newer versions of interpreters.

4 CONCLUSIONS

In this paper we have presented our work on something that we see as a substrate for model-driven creations. Resourcebus is designed to be as simple as possible, but to provide all foundational services that all modeling tools need to provide, such as model storage, versioning, access control, intelligent caching, and a way to plug-in model interpreters. Resourcebus aims to be compatible with current standards such as HTTP and RDF, and future standards such as Linked Data Platform (LDP). This way it could serve as the basis for sharing and distribution of models and metamodels on the web.

Our further work will be focused on finishing version 1.0 with enough examples so that other researchers would be able to test and potentially use Resourcebus for implementation of their own ideas. Issues that are still unresolved include combining multiple types of model storage including in-memory storage, finalization of the REST application style, implementation of more sample interpreters, such as ones for model-to-model transformations, and support for event processing.

REFERENCES

Beckett, D., Berners-Lee, T., Prud'hommeaux, E., Carothers, G., 2013. Turtle: Terse RDF Triple Language, W3C Candidate Recommendation.

Berners-Lee, T., 2006. Linked Data Design Issues. W3C-Internal Document, http://www.w3.org/DesignIssues/LinkedData.html.

Boag, S., Chamberlin, D., Fernández, M. F., et al, 2010. XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation.

Chacon, S., 2009. *Pro Git*. Apress, New York.

Crockford, D., 2008. *JavaScript: The Good Parts*. O'Reilly Media, Sebastopol.

Emerick, C., Carper, B., Grand, C., 2012. *Clojure Programming: Practical Lisp for the Java World*. O'Reilly Media, Sebastopol.

Erdweg, S., van der Storm, T., Völter, M., et al, 2013. The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge. In *Proceedings of the Software Language Engineering conference* (Indianapolis, USA, October 26-28, 2013). Springer.

Eysholdt, M., Behrens, H., 2010. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307-309.

Fielding, R., Taylor, R., 2002. Principled Design of the Modern Web Architecture, *ACM Transactions on*

Internet Technology (TOIT), Volume 2 Issue 2, May 2002.

Forsythe, C., 2013. Instant FreeetMarker Starter. Packt Publishing, Birmingham.

Geudens, T., 2012. *Resource-Oriented Computing with NetKernel: Taking REST Ideas to the Next Level,* O'Reilly Media.

Kay, M., 2007. XSL Transformations (XSLT) Version 2.0. W3C Recommendation.

Klyne, G., Carroll, J. J., 2004. Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation.

Koenig, D., Glover, A., King, P., et al, 2007. *Groovy in Action*. Manning Publications Co., New York.

McCandless, M., Hatcher, E., Gospodnetic, O., 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., Stamford.

Odersky, M., Spoon, L., Venners, B., 2010. *Programming in Scala*, Second Edition. Artima Press, Walnut Creek.

Sauermann, L., Cyganiak, R., 2008. Cool URIs for the Semantic Web. W3C Interest Group Note.

Speicher, S., Arwe, J., Malhotra, J., 2013. Linked Data Platform 1.0. W3C Last Call Working Draft.

Strauch, Ch., 2011. NoSQL Databases. Lecture Selected Topics on Software-Technology Ultra-Large Scale Sites, Manuscript. Stuttgart Media University, http://www.christof-strauch.de/nosqldbs.pdf.

Tolvanen, J.-P., 2006. MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages. *OOPSLA'06 Companion*, pp. 690-691.

Voelter, M., 2011. Language and IDE Development, Modularization and Composition with MPS. In GTTSE 2011, LNCS. Springer.