Metamodeling vs Metaprogramming: A Case Study on Developing Client Libraries for REST APIs

Markus Scheidgen[1], Sven Efftinge[2], and Frederik Martícke[1]

[1] Humboldt Universität zu Berlin, Germany
{scheidge,martícke}@informatik.hu-berlin.de
[2] Typefox GmbH, Kiel, Germany
sven.efftinge@typefox.de

Abstract. Web-services with REST APIs comprise the majority of the programmable web. To access these APIs more safely and conveniently, language specific client libraries can hide REST details behind regular programming language idioms. Manually building such libraries is straightforward, but tedious and error prone. Fortunately, model-based development provides different methods to automate their development. In this paper, we present our experiences with two opposing approaches to describe existing REST APIs and to generate type-safe client side Java libraries from these descriptions. First, we use an EMF-metamodel and a code generator (external DSL). Secondly, we use the Java compatible language Xtend and its metaprogramming mechanism *active annotations*, which allows us to alter the semantics of existing Xtend constructs to describe REST APIs within Xtend (internal DSL). Furthermore, we present related approaches and discuss our findings comparatively.

1 Introduction

Many of today's data intensive web applications (e.g. most social networks: Google+, Facebook, Twitter, etc.) provide access to their data via REST APIs. The representational state transfer (REST) principles impose very little restrictions on the development of clients. Ubiquitous technologies like HTTP and JSON facilitate development in almost all programming environments. On the downside, this technology combination provides little safety. IDEs cannot determine whether a certain request is part of the used API, if a certain parameter actually exists, if arguments have the right type, or whether the response is structured as expected.

Used to the safety and comfort (e.g. code-completion) of type-safe languages and IDEs, many developers build language specific client libraries for existing REST APIs. Those libraries transparently hide the necessary HTTP and JSON processing behind type-safe programming language idioms. The development of such client libraries is a straightforward deduction of boilerplate code from API documentation. This tedious and error prone process presents an archetypical use-case for model-based development (MBD).

Following our goal to create a set of homogeneous Java libraries for social networks and their existing REST APIs, we experimented with a metamodel- and a metaprogramming-based approach (implementations are provided as an open-source Github project [15]). First, we developed a metamodel that allows developers to describe REST APIs on a high level of abstraction through the use of domain (REST) specific modeling concepts (external DSL [9]). A code generator then produces the desired libraries from these descriptions.

Secondly, we use the Java compatible language Xtend [4] to describe REST APIs (internal DSL [9]). Xtend provides a metaprogramming concept called *active annotations*. Active annotations allow developers to eXtend the Xtend compiler with specialized semantics for existing language constructs. We use this to derive client libraries from REST APIs described with annotated Xtend classes and fields.

The next section will give a short introduction to REST and defines problems that we observed while developing Java libraries for existing REST APIs. In sections 3 and 4, we describe the two approaches and how we used them. We complement our approaches with those found in related work in section 5 and finally compare everything and draw conclusions in the closing section 6.

2 REST APIs and Client Libraries

Representational state transfer (REST, often also RESTful or ReST) is a set of principles for Internet client-server architectures originally formulated by Fielding [8]. REST is tightly associated with HTTP as the client-server-communication protocol. In fact, it is believed, that Fielding, who was part of the HTTP 1.0 and 1.1 standardization process, actively aligned HTTP with his REST principle [18]. There are two REST principles that are important for clients: *communication is stateless* and there has to be a *uniform interface*. The first principle, stateless communication, means everything necessary to understand a request and a respective response has to be part of the corresponding message. This facilitates simple and scalable server-side application design and is probably the most important factor for its wide adoption. In fact, 62% of APIs listed on *programmableweb.com* are REST APIs, compared to mere 17% of SOAP APIs. The second principle (uniform interface) can be realized at different levels according to Richardson's maturity model [2]. Level 0, there is no interface, just a black-box (no REST or RESTless); level 1, clients can identify specific resources on a server (i.e. through URLs); level 2, clients can also use different verbs (i.e. HTTP methods) to create, read, update, and delete (CRUD) resources; level 3, there is *hypermedia as the engine of application state* (HATEOAS). This means that resources reference each other with hyperlinks and that clients do not need to understand the full interface (i.e. resource URL path and parameters) because they simply navigate from one resource to another through contained hyperlinks. Although, HTTP allows different MIME-types, the majority of APIs use JSON to represent data. Fig. 1 shows an example of REST communication.

Fig. 1. Example request and response for a tweet search with Twitter's REST API. The resulting JSON is simplified; strings are abbreviated.

There are few restrictions for the development of REST clients: HTTP-, URL-, and JSON libraries exist for most programming environments; the learning curve compared to other web-service architectures, i.e. SOAP is low. But, client code becomes very verbose, since HTTP requests need to be constructed, possible errors handled, JSON needs to be parsed, etc. Furthermore, as stated earlier, there is no safety when compared to other architectures and to what most statically type-safe programming languages have to offer. Not surprisingly language specific and type-safe client libraries are in high demand. The left side of Fig. 1 shows a type-safe Java idiom that could be used to process the example request on the right side.

While the development of client libraries is pretty straightforward, we experienced a set of issues that any MBD of such libraries needs to cope with:

1. To avoid confusion, client libraries need to use the names that the original API uses. This includes names to distinct different kinds of resources in URL paths, names for URL parameters, and keys in JSON objects. But often, the existing names are not suitable for a particular programming language. Names might break naming conventions, inconveniently or illegally hide or override other names (e.g. `Class` in Java), or are outright forbidden (e.g. keywords).
2. JSON has a limited set of primitive types: boolean, strings, and numbers. This leads to non-trivial data mappings. For example, clients want to process dates as dates and not as strings that represent dates. Other examples

are URLs, colors, or GPS-coordinates. Furthermore, each REST API might encode the same data differently.
3. HATEOAS is not adopted by all services or often necessary links are missing. In practice, one needs to understand the URL part of a particular API. Furthermore, there are few conventions on how to organize resources and structure their URLs or their data. Even very similar concepts might be realized very differently in two APIs. One example is pagination (i.e. providing long lists of data over multiple requests). Some APIs provide the URLs to the next chunk of data (HATEOAS), some provide only partial URLs, some provide ids, other require the use of parameters with specific semantics.

3 Metamodeling

3.1 Metamodels and Code-Generation

The goal of a metamodel is to structurally define the usable constructs (abstract syntax) of a language. Like all object-oriented metamodeling languages, Ecore provides a combination of classifier (classes and data types) and feature (attributes and references) concepts to describes constructs (classes) and their possible attributes and relationships (features). Thereby, the metamodel is not concerned with what language instances (models) mean or what the language constructs are used for.

In order to give semantics to a metamodel, one can provide a code generator: a program that takes a metamodel instance (model) as input and translates it into code (i.e. an instance of a programming language). Thereby, a code generator realizes rules; each rule determines how to translate a certain language construct and each rule gives a meaning to a metamodel class and its features. While the code generator depends on a given metamodel, the metamodel is fully independent from the code generator. In fact, one can use multiple code generators that realize different semantics for the same metamodel. This is a major factor in MBD, since it facilitates the (re-)use the same model for different things.

In order to allow language users to express themselves with the defined constructs, a concrete notation (concrete syntax) and an accompanying tool (e.g. editor) are necessary. A tuple of metamodel, code generator, and notation/editor is often referred to as an external domain specific language (external DSL).

3.2 A Metamodel for REST APIs

Fig. 2 shows a simplified version of our REST API metamodel. The model contains the constructs necessary to describe all aspects of a REST APIs that are necessary to generate client libraries. The top part of the metamodel comprises core object-oriented constructs that we also find in many other languages (including UML, Java, or metamodeling itself). `Classes` can contain `Features` that describe slots for values of a certain `DataType`.

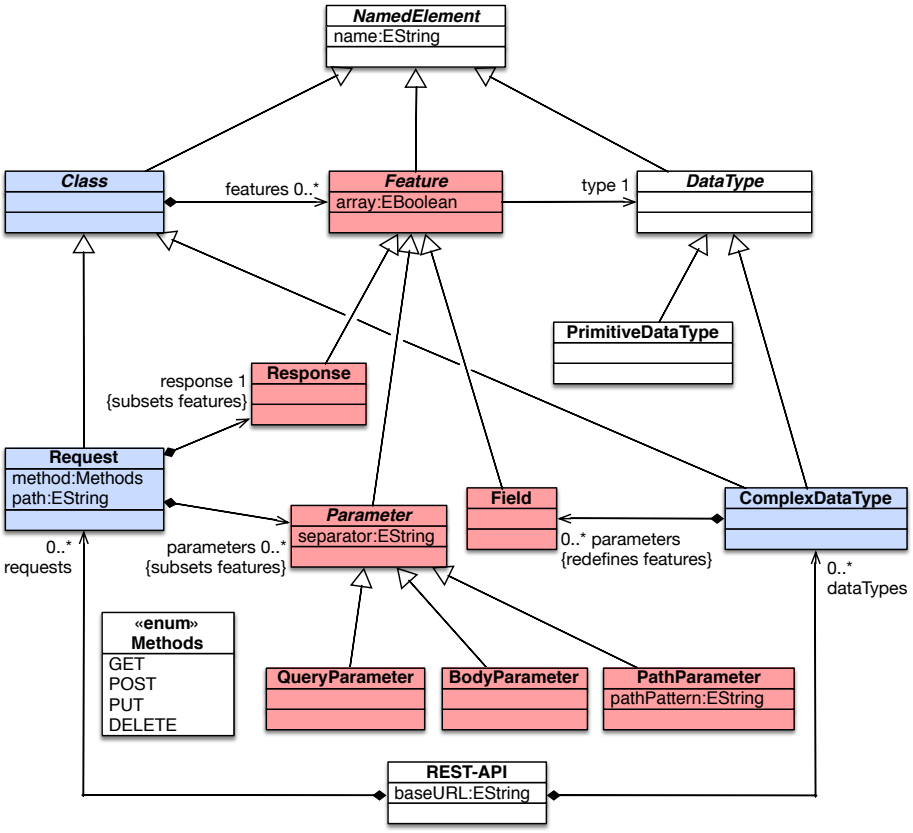

Fig. 2. Simplified metamodel for our REST client API generator.

There are two specializations of these common constructs in REST APIs: `Request` classes and `ComplexDataTypes`. A `Request` class defines a set of `Parameters` (communicated through either URL path, URL parameter, or HTTP body) and a specific `Response`. Both have respective types. A `ComplexDataType` (i.e. a schema for JSON objects) defines `Fields` (i.e. JSON key/value-pairs) which also have a respective type. Besides complex types, predefined and custom `PrimitiveDataTypes` can be used. All `Features` can also describe `arrays` of data.

We have written a code generator that translates REST API descriptions (metamodel instances) into client libraries (Java code). Instances of all class constructs (i.e. `Request` classes and `ComplexDataTypes`) are translated into Java classes, and all instances of feature constructs (i.e. `Parameter`, `Response`, `Field`) are translated into Java properties (i.e. pairs of get- and set-methods). Properties have Java types that match the respective `DataType` in the model.

Depending on the meta-class, some semantic variations are generated. A `Parameter` can only be modified before the response is accessed. Instead of properties, we generate a fluent interface for parameters. The response's get-method contains all the code that is needed to execute the request: construction and execution of the HTTP request with the corresponding HTTP verb and URL, waiting for its response, handling possible errors, parsing the JSON contained in the response, and instantiating the right Java class for the given `ComplexDataType`. Generated Java classes for `ComplexDataTypes` have a private field that holds a reference to the corresponding JSON object. The generated field access methods contain code that delegates calls to the wrapped JSON object and translates Java data to JSON data (set) and vice-versa (get). The code generator creates the right data-conversion and delegation code for arrays, other `ComplexDataTypes`, and the predefined primitive types. Users can declare custom conversion rules for their own primitive types (refer to 6 for more details).

3.3 Example

```
request SearchTweets path="search/tweets.json"
      response=SearchResults {
  query array q: String sperator=" "
  query since_id: long
  ...
}

data SearchResults {
  array statuses: Status
  search_metadata: SearchMetaData
}

data Status {
  text: String
  created_at: Date
  retweet_count: int
  user: User
  id: long
  ...
}

data User {
...
```

partial description of Twitter's API in a pseudo syntax

```
String searchKey = "ECMFA";

SearchResults results = Twitter
  .createSearchTweets()
  .q(searchKey)
  .xResult();

while (!results.getStatuses().isEmpty()) {
  for (Status status: results.getStatuses()) {
    System.out.println(status.getText());
  }

  results = Twitter.createSearchTweets
    .q(searchKey)
    .sinceId(
      results
        .getSearchMetadata()
        .getMaxId()
    ).xResult();
}
```

use of the generated wrapper API in Java

Fig. 3. Example API description and example use of generated library.

The left side of Fig. 3 exemplifies the use of the metamodel with a self explanatory concrete syntax. The example shows a description for a request class that defines the Twitter API's tweet search function that was depicted by example in Fig. 1. The description contains the appropriate path, a few parameters, and refers to a complex data type to wrap the expected response. The right side of the same figure shows how one can use the library that was generated from

this partial API description. The fluent interface for request creation allows to create and configure request in a single line. The method `xResult` triggers request execution and returns the result contained in the request's response. Note that the the response (as all features) has the proper Java type and the Java tooling can actually validate the proper use of these types. The rest of the example shows how to access the complex result data. You also see, how we can configure further request in typical REST fashion by passing state data from one response to the next request.

4 Metaprogramming

4.1 Xtend and Active Annotations

As a general paradigm, metaprograms are programs that transform the code of other programs or themselves into regular programs (object-programs) [16]. A specific approach to metaprogramming that Sheard [16] would classify as a homogeneous, manually annotated two staged metaprogram, is a metaprogram that uses annotations to replace idioms in the program with generated code in order to create object-programs. Readers unfamiliar with this application of metaprogramming can also think of this as a profile mechanism for programming languages, where one can define stereotypes (i.e. annotations) to assign a specialized meaning to existing language constructs.

Active annotations in Xtend provide such a metaprogramming mechanism. Xtend compiles into Java code and active annotations allow developers to inject specialized semantics into the compilation process. Syntactically, active annotations work like regular Java annotations. Semantically, Xtend provides a callback interface that hooks client code into the Xtend compiler. Developers can provide their own static semantics rules and provide further checks for the annotated elements. They can provide model-to-model transformation rules that expand the annotated elements into more complex Java structures. Finally, developers can provide code generator rules to implement the expanded annotated elements. In metaprogramming terms [16], implementations of the mentioned callback interface provide metaprograms that replace parts of a given object-program with different object-program code. The active annotations function as manual annotations to trigger the metaprograms for the intended object-program parts. The translation process only has two stages, so the generated code cannot contain further active annotations (or at least Xtend wont compile them as such). We have homogeneous metaprograms, since callback interface implementations are also Xtend programs.

4.2 A Set of Active Annotations to Describe REST APIs

As part of the REST API metamodel in section 3, we observed that request classes and complex data objects have a class/feature/datatype structure. The same structure that Java classes and their member fields also have. We can use

Fig. 4. Xtend annotation for REST API description, example use of annotations and example use of the generated library.

this similarity here. With active annotations, we cannot define new meta-classes; we cannot introduce new language constructs for requests or complex data types. But, we can use the existing Java/Xtend class and member field constructs to model request classes and complex data types. More concretely, we eXtend the semantics of classes and their members fields with active annotations.

The left side of Fig. 4 shows some of our annotation definitions. They are themselves annotated with annotations that describe details about the new annotations. The annotation `Active` turns a regular Java annotation into an active annotation. It also assigns the intended implementation of the before mentioned callback interface (i.e. the semantics of the annotation). We have defined two active annotations, one for request classes, called `Request` and one for complex data types, called `JSON`. When these annotations are used, they have to carry very little information, since most of the necessary information is already con-

veyed by the Xtend classes and their member fields. Classes already have a name, a set of member fields, and each field has a name as well as a type. The `Request` annotation has attributes for the URL path and for its response type. To denote specific characteristics of features, we add additional annotations. Fist, `Body-` and `PathParameter` to denote fields as body or path parameters (query parameter is the default). Secondly, `Name` to provide different feature names in case the necessary name is not a valid Java identifier (details in section 6). Thirdly, we have an annotation to describe the response (it's type or whether it is an array). Fourthly, to further describe complex data types, we can assign different names to fields or add data conversion to fields (also refer to section 6).

Semantically, the active annotations work similar to the code generator in our metamodel-based approach. The difference is that corresponding Java classes and fields already exist. However, the active annotations will replace the declared member fields with get- and set-methods (properties) and implement these methods similar to the described code generator in 3.

4.3 Example

We use the example from Fig. 1 again (upper right in Fig. 4). If you compare the Xtend code used to describe the API with the metamodel-based description in Fig. 3, you notice a strong and not surprising resemblance. Both examples describe the same part of the same API. The only difference is that in Xtend we cannot introduce our own keywords and have to use our annotations instead.

The example Xtend code that uses the generated library in the lower right of the same figure also shows strong resemblance to its Java counterpart (right of Fig. 3). In fact, the generated APIs are almost identical. But, since Xtend allows to access get-/set-methods like fields, omits empty parenthesis, and offers closures, we can express the same in a more compact form.

5 Related Work

Izquierdo et al. [11], Cánovas et al. [3], and Menkundle et al. [13] infer explicit schemas (e.g. in the form of Ecore-models) from the implicit schemas contained in example JSON data. These inferred descriptions are similar to our metamodel-based descriptions of complex data types. There is similar work based on metaprogramming side. The active Xtend annotation `Jsonized` [6] takes example JSON data and turns it into wrapper classes. `JsonProvider` [1] applies the same concept to type providers in F#.

Ed-Douibi et al. [5] build a server side REST framework for EMF-data. In a certain way, this can be interpreted as the opposite to our work. Instead of creating model-based descriptions of existing REST APIs, Ecore-models are interpreted as descriptions for new REST APIs. Gerhart et al. [10] is another example that represents EMF-data with JSON. With both works, JSON data can be processed safely via Java APIs generated from Ecore-models. But, in both

cases the mapping between implicit JSON schema and Java API is fix and you cannot use unconventional names or custom representations of primitive data.

There are also metamodel-based frameworks for generating server-side code for new REST APIs [17,14,12]. The used metamodels are all similar to each other and ours and a single description could be used for both server and client code. Unfortunately, most service providers have not adopted such approaches yet, and no formal descriptions of the examined APIs exist.

Our idea of metaprogramming with active annotations is only one way; reflection is another: Jackson [7] uses regular Java annotations and runtime-introspection to facilitate type-safe JSON data bindings. Type providers in F# are a different metaprogramming mechanism from the .NET world.

6 Discussion and Conclusions

When comparing the two presented approaches metamodeling and metaprogramming the general known arguments from comparing external with internal DSLs apply [9]: no syntactical limitations, closed language, specialized tools for external DSLs and language integration, existing/stronger tool support, and easier to (co-)evolve for internal DSLs.

Based on REST APIs as a specific application, further arguments in favor of external DSLs can be made. First, metamodeling is programming language independent and code generators for multiple languages and with different semantics can be used. In principle, we could derive client libraries for multiple languages, server-side code, and API documentation from the same description. Secondly, metamodels can define standards and metamodel-based descriptions can be used interoperably by different parties. During this work, we often wished that web-service providers would publish their APIs in such a formal manner. Even with an MBD-based description language for client libraries, we still have to gather descriptions from informal artifacts like API documentation. For some vendors, we even decided to develop specialized HTML-parsers to automatically translate online API documentation. The work of Izquierdo [11], Cánovas [3], and Menkudle [13] presents a different approach by inferring explicit schema information from actual application data.

In favor of internal DSLs, we can state that a metaprogramming mechanism like active annotations in Xtend drastically eXtends the possibilities to use a host language more specifically and elevates Xtend internal DSLs further from just being regular libraries. Now, we can introduce different static and dynamic semantics to existing language constructs without having to provide our own language tools. This is particularly ideal for applications that require to describe structures that consist of classes, features, and types like REST APIs, because similar concepts already exist in object-oriented host languages.

We stated three problems that arise when creating client libraries for one or multiple REST APIs in section 2. First, not all names are allowed in all programming languages. In a metamodel-based description, we can use all names, since the metamodel is programming language independent. The code generator

however has to identify non conforming names and has to change them accordingly. In an annotation based description, the names we want to use are already the names of Java/Xtend classes and fields. Consequently, we cannot always use the indented name. In general, we can also use implicit name conversions, but without further information, this can only be the same for all names (e.g. we could always convert camelCase names to snake_case names). This does not work for all non conforming names. Therefore, we often need to provide two names: one for the Xtend class or member, and one that is used to communicate with the server. We can provide the latter one with an annotation (example in Fig. 4 `id_str` vs `id`). In summary, we can solve this in both approaches, but for metaprogramming the description is less concise.

Secondly, programming language types are more specific than JSON types. Does a JSON string just mean a JSON string, or something that has a more specific type in a programming language, like a date or a color. Since this lays in the semantics of what is described and not within the syntax of it, it also does not matter whether we describe the API based on a metamodel or a set of annotations. In both cases, we have to provide conversion rules for features with primitive types. We allow users to program such conversions by implementing a given interface (refer to Fig. 4, left side and the end). In the metamodel, we can use an additional field in `Feature` to enable users to refer to converters by name; in an annotation-based description, developers can use an additional annotation `WithConverter` to add a specific converter to a feature (as done in Fig. 4: `created_at`). In Xtend, this referencing has good tool support, because we simply refer to a converter class written in Xtend/Java. With an external DSL, the same level of tool support is much harder to achieve.

Thirdly, different APIs implement common concepts differently. This problem lays within the semantics of the web applications themselves and has nothing to do with their APIs. REST APIs just provide a common way to access data, how a web-service decides to organize and structure this data is beyond the interface. Consequently, we can't do anything about it by means of describing REST APIs, neither with metamodels nor annotations. To identify common concepts and homogenize different APIs, we would need to create a common API and translate calls to the common API into calls for the individual specific existing APIs. This is a different topic and is beyond the scope of this paper.

Due to space restrictions, we had to leave out several aspects. We could not explain the use of constraints for request parameters and inheritance for complex data types (i.e. JSON data). The former is useful to enforce semantic conformance with APIs. The latter also for more concise descriptions. Rate-limits, authentication, and authorization are particularly challenging to integrate into client libraries homogeneously.

To summarize, both approaches, in principle, allow to describe REST APIs and allow to generate client libraries. The problems we identified are either solved in a similar fashion and cannot be solved by both approaches. In the end, general considerations, similar to those of external vs internal DSLs, decide. If you need to create libraries for a range of programming languages, or even want

to create server side code, you should opt for a metamodel-based approach. If the goal is to quickly develop a library for the Java world, Xtend and the presented active annotations allow to develop this API quickly and with existing tool support. In general, our experiences have shown that active annotations provide a meaningful addition to an internal DSL host language that allows to apply semantic variations to existing language constructs.

References

1. F# data: Json type provider, http://fsharp.github.io/FSharp.Data/library/JsonProvider.html
2. Betten, S.: Richardson maturity model. Tech. rep., Fachgebiet Software Engineering, Universität Hannover (2011)
3. Cánovas Izquierdo, J.L., Cabot, J.: Discovering implicit schemas in JSON data. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) LNCS(7977), 68–83 (2013)
4. Eclipse.org: Xtend, http://www.eclipse.org/xtend/
5. Ed-Douibi, H., Izquierdo, J.L.C., Gómez, A., Tisi, M., Cabot, J.: EMF-REST: generation of restful apis from models. CoRR (2015)
6. Efftinge, S.: Jsonized, http://github.com/svenefftinge/jsonized
7. FasterXML: Jackson json processor wiki, http://wiki.fasterxml.com/JacksonHome
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000)
9. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn. (2010)
10. Gerhart, M., Bayer, J., Höfner, J.M., Boger, M.: Approach to Define Highly Scalable Metamodels Based on JSON. BigMDE 2015 p. 11 (2015)
11. Izquierdo, J.L.C., Cabot, J.: Composing JSON-Based Web APIs. Web Engineering, ICWE 2014 8541, 390–399 (2014)
12. Maximilien, E.M., Wilkinson, H., Desai, N., Tai, S.: A Domain-Specific Language for Web APIs and Services Mashups. ServiceOriented Computing–ICSOC 4749, 13–26 (2007)
13. Menkudle, A., Sonawane, S., Jagtap, A.: Extracting Application Model from Restful Web Services for Client Stub Generation. International Journal of Computer Technology and Applications (IJCTA) 5(1), 226–232 (2014)
14. Rivero, J.M., Heil, S., Grigera, J., Gaedke, M., Rossi, G.: MockAPI: An agile approach supporting api-first web application development. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7977 LNCS, 7–21 (2013)
15. Scheidgen, M.: XRAW–Easy development of REST API client libraries with Xtend, http://github.com/markus1978/xraw
16. Sheard, T.: Accomplishments and Research Challenges in Meta-Programming. In: Taha, W. (ed.) Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG). pp. 2–44. Springer, Florence, Italy (2001)
17. Tavares, N., Vale, S., Luis, S., Brazil, M.a.: Towards Interoperability to the Implementation of RESTful Web Services : A Model Driven Approach. International Conference on Systems (ICONS) pp. 234–240 (2014)
18. Tilkov, S.: REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. dpunkt, Heidelberg (2009)