HeadREST: A Specification Language for RESTful APIs[⋆]

Vasco T. Vasconcelos[1][https://orcid.org/0000−0002−9539−8861], Francisco Martins[2][https://orcid.org/0000−0002−2379−7257], Antónia Lopes[1][https://orcid.org/0000−0002−2379−7257], and Nuno Burnay[1][https://orcid.org/0000−0001−6613−5192]

[1] LASIGE and Faculdade de Ciências, Universidade de Lisboa
[2] LASIGE and Universidade dos Açores

Abstract. Representational State Transfer (REST), an architectural style providing an abstract model of the web, is by far the most popular platform to build web applications. Developing such applications require well-documented interfaces. However, and despite important initiatives such as the Open API Specification, the support for interface description is currently quite limited, focusing essentially on simple syntactic aspects. In this paper we present HeadREST, a dependently-typed language that allows describing semantic aspects of interfaces in a style reminiscent of Hoare triples.

Keywords: REST · Web services · Description language

1 Introduction

Software services are not just a mechanism to compose software functionalities, but, in the present case, it was also the motto to bring together once again two groups of researchers, notably De Nicolas's and Vasconcelos' teams.

It all restarted in 2005, under the auspices of Sensoria, Software Engineering for Service-oriented Overlay Computers [15], a project revolving around the idea of service as a basis for service-oriented computing. In 2006 we authored together "SCC: A Service Centered Calculus" [3], a paper that laid down the foundations for describing the dynamic behaviour of services in terms of a process calculus. SCC introduces the notions of *service definition*, which provides for service behaviours, and of *service invocation*, which consumes instances of services. The communication between both ends of a service interaction happens in the context of a *session*. Inside this, processes send and receive messages isolated from other ongoing service interactions. A system is the parallel composition of service definitions, invocations, and ongoing sessions.

Following to this work, we concentrated on the problems of composing and orchestrating services, introducing SSCC [11]. This new calculus puts forward

[⋆] An early version of this paper was presented at the 24th International Conference on Types for Proofs and Programs, in June 2018.

a *stream* construct to play the role of a service orchestrator. In the following year, De Nicola and his team proposed CaSPiS that also features intra- and inter-session communication by using *streams* and *pipelines* [4]. CaSPiS further allows for reasoning about session cancellation and termination, scenarios in which processes may abandon or terminate their current sessions.

The explosive growth of the Web, and the adoption of services as one of the pillars for building distributed applications over the Web, continued to draw our attention to service-oriented computing. This time we decided to focus on RESTful web services. Confident is a research project on the formal description of RESTful web services using type technology [5].

Following the original spirit of REST [7], and in stark contrast to the philosophy of SOAP [9], state of the art service description systems use mainly natural language. While these descriptions may occasionally suit programmers, they are not adequate for machine consumption. Machine checkable service descriptions lie at the basis of static verification of RESTful-based applications, help in enforcing service fidelity, and in the construction and evolution of complex distributed applications.

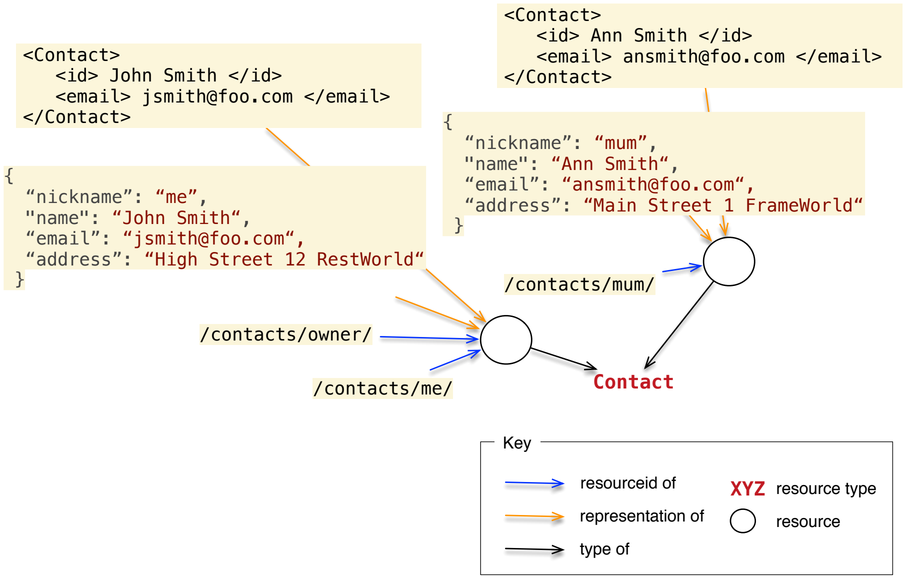
2 Context and related work

Representation State Transfer (REST) is an architectural style proposed as an abstract model of the web architecture. At its core lies the concept of *resource* [7]. According to Fielding and Taylor, a resource is a temporally varying membership function $M_R(t)$, mapping time t to a set of entities which are deemed equivalent [8]. The entities in the set $M_R(t)$ are *resource representations* and *resource identifiers*. REST uses a resource identifier to identify the particular resource involved in an interaction between components. Representations of resources are transferred between components in REST interactions; components perform actions on a resource by using a representation to capture the current or intended state of that resource.

In our running example—a simple contact management system—contacts are resources that admit (among others) a representation defined in terms of a nickname, a name, an email address, and a postal address. Figure 2 shows an example of two contacts. One of the contacts bears two different identifiers: `me` and `owner` (the owner of all contacts). Both contacts have JSON and XML representations that also differ in the amount of information included.

Systems that conform to the constraints of the REST architecture are called *RESTful*. A RESTful system can be seen as a set of resources together with the actions that can be performed on these. A RESTful API can be abstracted as a set of resource identifiers together with the actions that can be performed on each resource via that identifier.

REST systems typically communicate over HTTP and interface with external systems as web resources identified by URIs. The actions in this case include GET, POST, PUT, DELETE. In systems that communicate over HTTP, additional information can be sent in the *request* for the execution of the action. This



```
<Contact>
   <id> John Smith </id>
   <email> jsmith@foo.com </email>
</Contact>
```

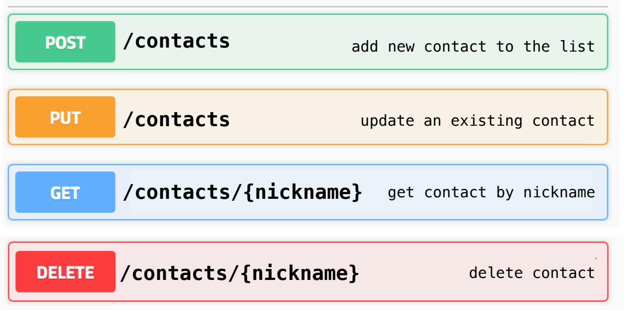
```
{
  “nickname”: “me”,
  "name": “John Smith“,
  “email”: “jsmith@foo.com“,
  “address”: “High Street 12 RestWorld“
 }
```

```
<Contact>
   <id> Ann Smith </id>
   <email> ansmith@foo.com </email>
</Contact>
```

```
{
  “nickname”: “mum”,
  "name": “Ann Smith“,
  “email”: “ansmith@foo.com“,
  “address”: “Main Street 1 FrameWorld“
 }
```

Fig. 1. Two resources in a contact management REST service.

comes in the form of parameters embedded in the URL, headers, and body. Results always include a *response*. The table below shows four actions in the contact management system, together with their URIs and a textual description.



POST	/contacts	add new contact to the list
PUT	/contacts	update an existing contact
GET	/contacts/{nickname}	get contact by nickname
DELETE	/contacts/{nickname}	delete contact

Different *interface description languages* (IDLs) have been purposely designed to support the formal description of REST APIs. The most representative ones are probably Open API Specification [12] (originally called Swagger), the RESTful API Modeling Language [13] (RAML), and API Blueprint [1]. These IDLs allow a detailed description of the syntactic aspects of the data transferred in REST interactions and are associated to a large number of tools, in particular for documentation generation, client code generation in different programming languages, and for test generation. Focused on the structure of the data exchanged, they ignore important semantic aspects, such as the ability to relate different parts of the same data, to relate the input against the state of the service, and to relate the output against the input. For instance, in the case of the contact management system, none of IDLs discussed here allow expressing facts such as that, in the creation of a new contact, the nickname must be shorter than the full name or that the name should be unique across all names known to the system. Similarly, these languages do not allow expressing that the type of representation transmitted in the response to a GET action depends on the value of a given query parameter.

3 HeadREST

Our approach to the description of RESTful APIs relies on two key ideas:

- *Types* to express properties of server states and of data exchanged in client-server interactions and
- *Pre- and post-conditions* to express the relationship between data sent in requests and that obtained in responses, as well as the resulting state changes in servers.

These ideas are embodied in HeadREST, a language built on the two fundamental concepts of DMinor [2]:

- *Refinement types*, `x:T where e`, consisting of values `x` of type `T` that satisfy property `e` and
- A *predicate*, `e in T`, which returns true or false depending on whether the value of expression `e` is or is not of type `T`.

HeadREST allows to describe properties of data and to observe state changes in server through a collection of *assertions*. Assertions take the form of Hoare triples [10] and are of the form

$$\{\phi\}\ (a\ t)\ \{\psi\}$$

where a is an action (`GET`, `POST`, `PUT`, or `DELETE`), t is an *URI template* (e.g., `/contacts/{i}`), and ϕ and ψ are boolean expressions. Formula ϕ, called the *precondition*, addresses the state in which the action is performed as well as the data transmitted in the request, whereas ψ, the *postcondition*, addresses the state resulting from the execution of the action together with the values transmitted in the response. The assertion reads

> If a request for the execution of action a over an expansion of URI template t carries data satisfying formula ϕ and the action is performed in a state satisfying ϕ, then the data transmitted in the response satisfies formula ψ and so does the state resulting from the execution of the action.

A simple contact management system includes different (abstract) resources, which HeadREST captures as *new types*. Resources are introduced as follows.

```
resource Contact
```

Each resource may be associated to zero or more *representations*, each of which is given a particular type. The type system of HeadREST is structural, yet the language provides for *type abbreviations* in order to ease the writing of complex API descriptions. The syntax below introduces an identifier (`NameAndEmail`) for an object type, intended to represent resource `Contact`. `NameAndEmail` is an object composed of a `name` (a string of 3–15 lower and uppercase letter) and an `email` (a string containing the symbol `@`).

```
type NameAndEmail = {
  name: (x: string where matches(x, ^[a-zA-Z]{3,15}$)),
  email: (x: string where contains(x, "@"))
}
```

Equipped with the declaration of a new resource (`Contact`) and a name for one of the representations of the resource (`NameAndEmail`), one can write a few assertions describing the behaviour of the API. One that describes a successful contact creation could be written as

```
{request in {body: NameAndEmail} &&
 ∀c:Contact. ∀r:NameAndEmail.
   r repof c ⇒ request.body.name ≠ r.name
}
POST /contacts
{response.code == 200 &&
 response in {body: NameAndEmail, header: {Location: URI}} &&
 request.body == response.body &&
 ∃c:Contact. response.body repof c &&
             response.header.Location uriof c
}
```

where `request` and `response` are builtin identifiers, and predicates `repof` and `uriof` describe values associated to resources as described in Section 2 (cf., Figure 2).

The precondition first establishes that `request` contains a field named `body` of type `NameAndEmail`, and then asks the new contact `name` (provided in the `body` of the `request`) to be unique across all contacts and their representations, hence the double quantification (first on resources and then on their representations). In such a case, the postcondition signals success (`code` 200) and states that `response` includes a representation (in field `body`) that is exactly what was sent in the request. Furthermore, the response includes an URI (in field `header.Location`) of the newly created `Contact` resource `c`.

A different assertion for the same pair action-URI describes the *conflict* story: if the name of the new contact is known to the server, then this signals conflict (`code` 409).

```
{request in {body: NameAndEmail} &&
 ∃c:Contact. ∃r:NameAndEmail.
   r repof c ⇒ request.body.name == r.name
}
POST /contacts
{response.code == 409}
```

We have used HeadREST to describe different APIs, including a part of GitLab (800 lines of spec code). We have developed an Eclipse plugin to validate the good formation of HeadREST specifications [5], a tool to automatically test REST APIs against specifications [6], and a tool to generate server stubs and client SDKs from HeadREST specifications [14].

4 Conclusion

In this short abstract we informally present HeadREST, a language designed to support the entire application lifecycle based on REST APIs. We briefly discuss the language via a very simple example that illustrates the challenges of describing REST APIs and the expressiveness of our specification language.

Equipped with such an API description, we build tools that (a) validate the good formation of HeadREST specifications, (b) generate server stubs and client SDKs from HeadREST specifications, and (c) that automatically test REST APIs against specifications.

We intend to explore the specification of security issues in REST context, in particular, how to use the HeadREST language to ensure compliance with authentication and confidentiality requirements.

Acknowledgments This work was supported by the Foundation for Science and Technology (FCT) through project CONFIDENT (PTDC/EEI-CTP/4503/2014) and the LASIGE research unit (UID/CEC/00408/2019).

References

1. API blueprint. https://apiblueprint.org/. Retrieved 7-Jan-2019.
2. Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012.
3. Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. SCC: A service centered calculus. In *Proceedgins of the Third Internation Conference on Web Services and Formal Methods (WS-FM)*, pages 38–57, 2006.
4. Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Caspis: a calculus of sessions, pipelines and services. *Mathematical Structures in Computer Science*, 25(3):666–709, 2015.
5. Confident, a toolchain for the construction and evolution of REST APIs. http://rss.di.fc.ul.pt/tools/confident. Retrieved 7-Jan-2019.
6. Fábio Ferreira. Automatic test generation for RESTful APIs. Master's thesis, Faculty of Sciences, University of Lisbon, 2017.

7. Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
8. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Techn.*, 2(2):115–150, 2002.
9. HTTP Working Group. SOAP: Simple object access protocol. https://tools.ietf.org/html/draft-box-http-soap-00. Retrieved 31-Jan-2019.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
11. Ivan Lanese, Francisco Martins, Vasco Thudichum Vasconcelos, and António Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proceeedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 305–314, 2007.
12. Open API Initiative. https://www.openapis.org. Retrieved 7-Jan-2019.
13. RESTful API modeling language. https://raml.org. Retrieved 7-Jan-2019.
14. Telmo Santos. Code generation for RESTful APIs in headREST. Master's thesis, Faculty of Sciences, University of Lisbon, 2018.
15. Sensoria, software engineering for service-oriented overlay computers. http://sensoria.fast.de/. Retrieved 31-Jan-2019.