Evaluating a Representational State Transfer (REST) Architecture

What is the impact of REST in my architecture?

Bruno Costa, Paulo F. Pires, Flávia C. Delicato
Department of Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
{brunocosta.dsn, paulo.f.pires, fdelicato}@gmail.com

Paulo Merson
Software Engineering Institute
Pittsburgh, PA, USA
pmerson@acm.org

***Abstract*— The use of Representational State Transfer (REST) as an architectural style for integrating services and applications brings several benefits, but also poses new challenges and risks. Particularly important among those risks are failures to effectively address quality attribute requirements such as security, reliability, and performance. An architecture evaluation early in the software life cycle can identify and help mitigate those risks. In this paper we present guidelines to assist architecture evaluation activities in REST-based systems. These guidelines can be systematically used in conjunction with scenario-based evaluation methods to reason about design considerations and trade-offs. This paper also presents a proof of concept to describe how to use the guidelines in the context of an Architecture Trade-off Analysis Method (ATAM) evaluation.**

Keywords- software architecture evaluation; scenario-based evaluation guidelines; REST

I. INTRODUCTION

Architectural decisions determine the ability of the implemented system to satisfy functional and quality attribute requirements (e.g., interoperability, security, and usability). Since these architectural decisions are among the first to be taken during a software development life cycle, and they affect later stages of the development process, the impact of architectural mistakes is high. Therefore, it is important to inspect the architecture early in the development process to identify and mitigate any risks of the software solution not satisfying the quality attribute requirements. This inspection activity is referred to as software architecture evaluation [1].

Many Web-based systems today use Representational State Transfer (REST). REST was originally proposed as a software architectural style for distributed systems in Roy Fieldings' PhD dissertation [4]. Fieldings presents a set of *REST constraints*, such as uniform interface, stateless, and client-server, which are design restrictions prescribed by the REST architectural style. By using REST, some quality attributes of the system, such as interoperability and modifiability, are positively impacted, whereas others, such as performance and reliability can be negatively impacted. Architecture evaluators should identify places and design decisions in the architecture that influence the ability of the system to meet the quality attribute requirements. This task requires a clear understanding of the trade-offs between quality attributes. The widespread use of REST has recently brought to the minds of project managers, architects and architecture evaluators the following question: "what is the impact of using REST in my software system?" [2]. Bianco and colleagues [3] wrote a report to aid architecture evaluators to answer this question and perform an efficient evaluation in the more general context of SOA. Although such report proved to be useful in an architecture evaluation of SOA-based systems, it does not provide in-depth analysis on the impact of REST constraints on quality attribute requirements.

The purpose of this paper is to provide guidance for architecture evaluation activities in systems that use the REST architectural style. We discuss several design aspects and provide guidelines on important inspection points. We also discuss how the architecture can be probed by the evaluation team and propose the types of questions that should be asked during the evaluation process. To demonstrate how the guidelines can help evaluators to mitigate risks in an architecture evaluation, we present a proof of concept project to describe how to use the guidelines in the context of ATAM (Architecture Trade-off Analysis Method) [5], a scenario-based evaluation method used to uncover risks and trade-offs reflected in architectural decisions relating to quality attribute requirements. The rest of this paper is organized as follows: Section II introduces scenario-based architecture evaluation. Section III describes the research protocol used to develop this work. In Section IV we present the architectural foundations of REST from the point of view of an architecture evaluator. REST general quality attribute scenarios are described in Section V. Design questions that affect quality attributes are discussed in Section VI. A proof of concept evaluation that uses the guidelines introduced in this paper is described in Section VII. Finally, Section VIII presents the conclusions and future work.

II. SCENARIO-BASED ARCHITECTURE EVALUATION

Scenario-based evaluation methods evaluate the software architecture's suitability according to a set of scenarios of interest. A *quality attribute scenario* is a structured description of a quality attribute requirement that is unambiguous and testable [1]. Quality attribute requirements can be expressed

by two types of scenarios: *(i) general quality attribute scenarios* (hereinafter *general scenarios*), which are generic and can be applied to any software system; and *(ii) concrete quality attribute scenarios* (or *concrete scenarios*), which are specific to the particular system under consideration [1].

The specification of a quality attribute scenario has six parts: *(1)* **Source of stimulus**: Some entity (a human, a computer system, or any other actuator) that generates a given stimulus; *(2)* **Stimulus**: A condition that requires a response when it arrives at a system; *(3)* **Environment**: The conditions for the stimulus occurrence; *(4)* **Artefact**: The artefact that is stimulated; *(5)* **Response**: The activity undertaken as the result of the arrival of the stimulus, and; *(6)* **Response measure**: The response of the system, which should be measurable in some fashion so that the requirement can be tested.

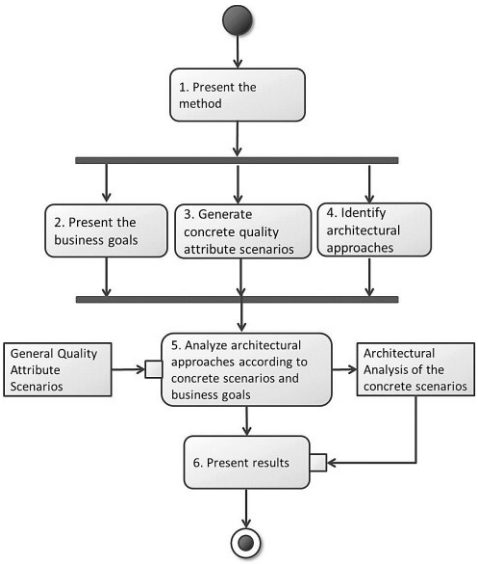

Fig. 1 - Typical activities in scenario-based evaluation methods

Scenario-based architecture evaluation methods typically include 6 activities. Some activities can be done in parallel, as shown in the UML activity diagram in Fig. 1. These activities are: *(1)* **Present the method**: The evaluation team presents a quick overview of the method steps, the techniques used, and the outputs of the process activities; *(2)* **Present the business goals**: The system manager briefly presents the business goals and context for the architecture; *(3)* **Generate quality attribute scenarios:** The architect, along with other stakeholders and the evaluation team identify the quality attribute requirements and describe them as six-part concrete scenarios, as described above. General scenarios can be used in this step as reference of concrete scenarios; *(4)* **Identify architectural approaches**: The architect presents an overview of the architecture and the evaluation team itemizes the architectural approaches (design decisions made by architects that can affect quality attribute requirements and business goals); *(5)* **Analyse the architectural approaches according to concrete scenarios and business goals**: The evaluation team probes the architectural approaches in light of the concrete scenarios and business goals, and then creates the architectural analysis of the concrete scenarios. In order to probe the architecture, the team uses design questions based on their practical experience and on architecture evaluation guides, and; *(6)* **Present results**: The evaluation team recapitulates the method steps, outputs, and presents the architectural analysis.

The main contribution of this paper is to propose a guidance that includes general scenarios (used in activity 3) and guidelines (used in activity 5) specific to REST-based solutions.

III. RESEARCH PROTOCOL

The guidance proposed in this paper was developed following the research protocol showed in Fig. 2. The following sections describe each activity of such protocol.

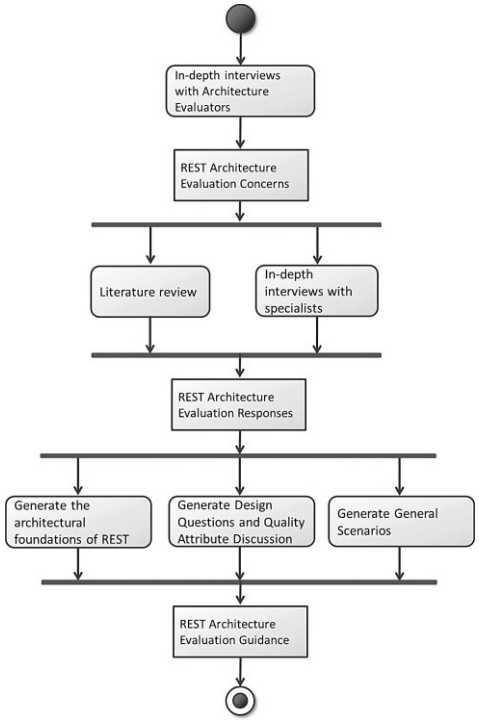

Fig. 2 - Activities in our research protocol

A. *In-depth Interviews with Architecture Evaluators*

The first activity of the protocol consisted of a set of in-depth interviews with architecture evaluators to gather an initial set of REST-specific architecture evaluation concerns from a practitioners' point of view. Following a systematic methodology [6], we interviewed two experts in architecture evaluation that have coordinated more than 50 architecture evaluations in industry using ATAM. With two architecture evaluation leaders interviewed, it was possible to compare and analyse their common needs. As the result of the interviews, the evaluators indicated that a REST architecture evaluation guide must address three main issues. In our protocol, these issues are called *REST Architecture Evaluation Concerns*:

EC1- "Explain the architectural foundations of REST from an architecture evaluator's point of view";

EC2- "Discuss quality attributes and general scenarios that benefit from REST";

EC3- "Discuss (in detail) how REST contributes to the quality attributes and where typical trade-offs are".

B. Literature Review

The next activity of our protocol was to carry on a comprehensive literature review on REST. In the last years, much information about REST has been published with considerable contributions. These contributions are particularly important to answer the *REST Architecture Evaluation Responses*. The literature review was executed according to the methodology proposed by Kasunic [7]. We started by searching for publications from major research databases of Computer Science, such as ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect using keywords such as *rest*, *representational state transfer*, *quality attributes*, and *design*.

In order to identify and extract answers for REST architecture evaluation concerns (EC1, EC2 and EC3), one researcher systematically analysed the title and abstract of 384 scientific articles published in journals and conference proceedings for the period 2000-2013. If it was unclear from the title and abstract whether an article discussed the answers for evaluation concerns, the entire article was read. After this step, 42 scientific papers were selected. In addition, 8 books and 9 technical reports were selected. The selected papers, books and technical reports were studied to find information useful to answer the concerns. Then, the information considered useful was grouped and summarized according to concerns EC1 (described in Section IV), EC2 (described in Section V), and EC3 (described in Section VI) in order to generate the guidance and to create the questionnaire used in the next activity.

C. Retrieving information from experts

We next engaged several practitioners and researches who have long been designing and studying REST solutions. Their experience about REST design was used as the second source of knowledge to respond the REST architecture evaluation concerns. The interaction with these experts also followed the methodology in [6]. First, we developed a questionnaire based on the performed literature review (activity 2). The questionnaire was made available as an online survey and the link sent to ten experts from industry and academia. All of them have worked with REST for more than 5 years and have experience with 5 to 10 different REST projects. The interviewers were: 3 researchers, 3 REST service providers, 2 consultants, and 2 REST service consumers. The researchers have published several papers, participated or organized conferences in the field, such as the International Workshops on RESTful Design (WS-REST). Providers work in mid to large companies from the TV, Internet movies, news media, and online maps domains. Their REST services have 10 to 30 million clients that use more than one thousand different kinds of devices. Consultants have worked on several REST lifecycle activities, such as conception, API exposing, and resources definition. Consumers are developers that have extensively used REST Web services from various providers. After applying the questionnaire and analysing the responses, we observed that the results were not insightful enough to be used as basis for architecture evaluation. The experts indicated to us that they did not manage to find the suggested 30 minutes required to answer the questionnaire. We believe this lack of time of the respondents was the main reason for the poor feedback. As a fall-back strategy, we decided to conduct in-depth interviews with the same 10 experts that we had sent the questionnaire, also motivated by our previous successful direct interaction with the architecture evaluators. The interviews with experts were also planned and executed according to the methodology proposed by Oishi [6]. In each interview we walked through the questions in the original questionnaire revisiting the interviewee's previous answers to those questions. We also added questions on the fly according to the answers.

D. REST Architecture Evaluation Responses

The literature review and in-depth interviews with experts produced a significant amount of data that was captured in interview datasheets, text documents, annotations, and literature excerpts. The next activity of the protocol was intended to organize all information in a structured document to generate the guidelines. Next sections show the result of our research, which is structured according to the REST architecture evaluation concerns.

IV. FOUNDATIONS OF REST FOR ARCHITECTURE EVALUATION (EVALUATION CONCERN EC1)

One of the main goals of an architecture evaluation is to inspect the risks to address attribute requirements in software architecture. The next sections describe the foundations of REST taking into account its impact in quality attributes. The foundations are based on interviews and literature review, primarily on [4], [9], [18], [19], [20], and [21].

A. REST Constraints

Representational State Transfer (REST) is a software architectural style for distributed systems [4]. Roy Fielding has described six constraints that define the REST style, each of which promotes a different set of quality attributes. REST can be described as (1).

$$REST = (C\text{-}S, S, \$, U, L, CoD) \quad (1)$$

Each variable in expression (1) is a constraint. Next, we briefly describe each constraint along with respective impact on quality attributes and important points that evaluators should inspect to assess that impact. In Section VII the constraints will be distilled into REST design questions.

1) Client-Server (C-S, S, $, U, L, CoD)

Client-server is a frequently found architectural style for network-based applications. This style describes distributed systems that involve separate clients that request services from server components over a network connection. In REST, requests are initiated by *user agents* (clients) and ultimately processed by an *origin server* (server), which provides services through a resource hierarchy. REST has a strong focus on decoupling client and server; however, several architectures show challenges to define which modules will act either as a client or as a server. The main benefits of the Client-Server style are separation of responsibilities, independent evolution and maintainability. Evaluators should inspect the definition of the boundary between client and server according to cohesion and the independent evolution of each one.

2) Stateless (C-S, S, $, U, L, CoD)

The stateless constraint describes that all information needed to understand the conversation state data between *origin servers* and *user agents* must be included in the request and response messages. That means that no conversational state is kept on the server between requests; state is kept on the client. Important inspection points are related to server state data flow. The Stateless constraint enables replication of servers and hence promotes availability, scalability and reliability; on the other hand, it may decrease performance due to the need for sending the conversational state data embedded in request and response messages.

3) Cache (C-S, S, $, U, L, CoD)

The cache constraint is added in order to improve performance. A cache element acts as a mediator between client and server. The objective is to avoid a request-response interaction when the information is present in the cache, thus avoiding network traffic. In multi-tier solutions (layered system constraint), a cache may also be present in intermediary tiers. Although cache is an ordinary feature of the World Wide Web, it is not always used in REST services that are part of SOA solutions. Evaluators should inspect what data has to be cacheable. The degree to which the cache will increase network efficiency and hence performance, depends on the cache strategy. However, the use of a cache may decrease reliability in cases when the client may consume stale data from the cache.

4) Uniform Interface (C-S, S, $, U, L, CoD)

Uniform Interface across components is the central feature that distinguishes REST from other network-based styles [4]. Uniform Interface is closely related to resources, identifiers and representations. A resource is any important concept in the system domain that we want to make accessible through a uniform interface. A service consumer addresses resources through a unique identifier (URI) and a representation. The representation of a resource refers to a hypermedia document that brings any useful information about the state of the resource and links to other associated resources. Finally, the representation of a resource must be accessed by a simple interface that defines an identification for the resource and common methods to access. This interface is the uniform interface. Uniform interface constraint impacts positively in interoperability and discoverability quality attributes.

5) Layered System (C-S, S, $, U, L, CoD)

The layered system constraint as described by Fielding is not exactly the more general layered pattern often found in the patterns literature. In fact, since 2000 the architectural pattern more easily identified with the REST layered constraint described by Fielding is *multi-tier*. Multi-tier is an architectural style that is a specialization of the client-server style [8], where an intermediary tier acts as server to the previous tier and as client to the subsequent tier. Fielding suggests that elements in a tier should not "see" beyond the next tier hence containing overall system complexity. Legacy systems, databases and backend systems in general are often enclosed in the rearmost tier. Elements in an intermediary tier can be load-balanced for improved availability, scalability, and system throughput. However, the deployment of components across several layers may increase the response time for a given request, due to the multi-hop communication across layers. Although simple REST services can be available on the "server-side" of client-server architectures, REST services are often found on "server-side" tiers of multi-tier applications developed using Java EE, .NET, or other implementation platform suitable for multi-tier applications.

6) Code-on-Demand (C-S, S, $, U, L, CoD)

Code-on-Demand is an optional constraint that enables a dynamic architecture where the user agent logic can be extended by code received from the service. Code can be for instance an applet or JavaScript function used in web pages to render widgets or validate user input. As another example, consider a client that obtains from the server the code that is required to run in order to encrypt message payloads. Code-on-demand is a form of runtime variability since the functionality and behaviour of user agent can vary at runtime per the code received in response messages to a REST service. *CoD* promotes extensibility. On one side, the basic client implementation devoid of any code received on demand is lightweight, but on the other side, the complexity of this client implementation increases to handle the runtime code additions. Interoperability may decrease since the downloaded code has to be compatible among all service consumers. Security is also a concern to prevent malicious code to reach the clients.

B. Pragmatic REST

The aforementioned constraints *(C-S, S, $, U, L, CoD)* represent what we call "Pure REST". They do not define a specific technology or platforms for implementing REST-based systems. REST has become a popular architectural style for designing and developing distributed systems, and technical solutions and development frameworks have been proposed for implementing REST services. We call these solutions "Pragmatic REST", because they focus on practical aspects of designing and implementing services that follow the REST architecture style.

Pragmatic REST uses Internet technologies, primarily http and media types. The REST constraints (C-S, S, $, U, L, CoD) can be applied over http and related technologies to implement web services that are called REST services or RESTful services [9]. Architects, developers, and architecture evaluators of REST-based solutions should know how the http protocol works. Http is primarily specified in RFC 2616 [10]. In Pragmatic REST, the most important constraint is Uniform Interface and its related concepts: Resources, Identifiers and Representations. That constraint distinguishes REST services from other web-based services, such as SOAP services.

V. EXAMPLES OF REST GENERAL QUALITY ATTRIBUTE SCENARIOS (EVALUATION CONCERN EC2)

A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders [1]. In this section we enumerate general quality attribute scenarios for ten quality attributes that are important to consider when using the REST style, according to the experts we interviewed (Section III). These general scenarios are referenced in the REST design questions

(Section VI) to illustrate how they are affected by design decisions. There is no universal definition for each quality attribute, several definitions and taxonomies of quality attributes are found in the literature [22, 23]. But that fact is not an issue in scenario-based architecture evaluations, because what actually defines the quality attribute requirement is the scenario statement. We label a scenario, say as a "usability scenario" or "performance scenario", merely for organization purposes.

TABLE I. GENERAL REST QUALITY ATTRIBUTE SCENARIOS

Quality Attribute	Scenario
Interoperability	**I1-** A service consumer 'A' requests a resource 'R1' and receives the representation of the actual state of 'R1' in response message. **I2-** A service consumer 'A' requests a resource 'R1' in a specific format (media type) and receives the representation of the actual state of 'R1' in response according to the requested format. **I3-** A service consumer 'A' requests a resource 'R1' and can understand all information presented in the response message.
Reliability	**R1-** A service consumer 'A' requests a resource 'R1' in a specific version specified directly in the URI and receives the representation of the actual state of 'R1' in response message. **R2-** Based on the domain model, a service consumer 'A' builds the URI of resource 'R1' and receives the representation of the actual state of 'R1' in response.
Security	**S1-** A service consumer 'A' with insufficient privileges requests confidential information to a service interface 'X', 'X' denies the request and informs 'A' about the lack of authorization. **S2-** An authenticated and authorized service consumer 'A' requests a confidential resource 'R1' which it has access to and receives the representation of the actual state of 'R1' in response.
Testability	**T1-** A developer wants to test a service. If an error occurs when processing the request, the service can be configured to provide in the response all information to identify the error, including the execution trace.
Performance	**P1-** A service consumer 'A' performs an action in a resource 'R1' and receives the representation of actual state of 'R1' in response in less than n milliseconds.
Availability	**Av1-** The web server where the REST services run is flooded with a number of requests N percent higher than normal and the services remain responsive.
Modifiability	**M1-** A developer modifies the core logic and internal data sources of a service, but the service contract (uniform interface, supported URIs and representations) remains the same; the effort to effect these changes is bound to N person-days. **M2-** The representation structure of a resource 'R1' and its relations to other resources change, and resource identification (URI) is not affected. **M3-** Resource representations are modified and the respective services must correctly process requests from service consumers that use the old version and consumers that use the new version of the resources under the same URI.
Safety	**Sa1-** A service consumer 'A' performs many requests by using the idempotent method http PUT and the resource has the same value as the first request performed. **Sa2-** A service consumer 'A' performs requests by using safe methods (such as http GET and OPTIONS) and the resource is not modified.
Discoverability	**D1-** A service consumer 'A' requests a resource 'R1' and receives the URIs to find resources associated to 'R1' inner the response message.
Functionality	**F1-** Based on the default organization of a resource identification in a service interface, a service consumer 'A' can compose a resource URI. **F2-** The developer of a service consumer finds clear and up-to-date documentation of the service interface. **F3-** A service consumer 'A' performs an action on a resource and receives its identification (URI) in the http header field Location. **F4-** A service consumer 'A' requests a representation of some attributes of a resource 'R1', defines the number n of pages and receives the actual state of attributes requested of 'R1' in n pages in response. **F5-** A service consumer 'A' wants to perform operations in a resource and 'A' can only use http primitives for that.

VI. REST DESIGN QUESTIONS THAT AFFECT QUALITY ATTRIBUTES (EVALUATION CONCERN EC3)

In architectures based on the REST style, several design decisions bear quality trade-offs and are driven by the quality attribute requirements. This section covers topics that are particularly relevant when designing REST-based systems: Design of Resources (subsection A); Representation and identification (subsection B); Documentation and testing (subsection C); Behaviour (subsection D), and; Security (subsection E). In each topic we present the design questions that evaluators should ask referencing to the general scenario (Section V) that is impacted.

A. Design of Resources

Design of Resources is related to the design of operations, capabilities, data elements and other resources that will be exposed in a service interface. When evaluating resource design, the following questions can help to identify risks:

1) What is the domain model of the application?

The domain model represents the knowledge about the domain of the application. It describes the various entities, their attributes, roles and relationships. The domain model can be represented, for example, by a UML class diagram. Evaluators should check if the domain model was validated by the application stakeholders.

2) What data will be exposed as resources?

Evaluators should have an overall understanding of what entities will be exposed as resources via REST services.

3) Are resource representations standardized within the entire application, department, or enterprise?

The representation of a given resource should be the same in terms of format (e.g., XML, JSON) and structure across all services that deal with that resource. Ideally, this standardization should be enterprise-wide, but minimally it should span the entire software system. If a given resource has different representations across the system, interoperability is severely impaired. Performance may also be negatively affected due to the need for data format transformations (e.g., JSON to XML) or data model transformation (e.g., XSLT transformation from one schema definition to another).

4) What types of service consumers will interact with the REST services?

This question should elicit the nature of the software components that will act as consumers to REST services. They can be web pages on a browser, a Java or .NET component running on a server machine, mobile apps, standalone applications, or any other kind of program that can communicate via http. We should also elicit the nature and properties of the communication channel. Are service consumers at the same local network or intranet where the REST service is? How fast and reliable is the connection? Is the communication channel encrypted?

5) Is confidential information exposed as a resource?

General scenario S1 describes general security issues that have to be analysed when designing resources. Therefore, evaluators should know whether data confidentiality is a requirement for any REST service.

B. Representation and identification

The following questions related to the representation and identification of resources can help to spot risks in the design of REST services:

1) What format is used to represent resources?

A resource is represented by a document with a media type identified by a name (such as text/xml), and declared in Content-Type http headers. XML (application/xml) is a very common representation format for resources, although other formats, such as HTML, ATOM, and JSON, are also widely used. Development frameworks and platforms provide different levels of support for different formats. The choice of representation format also affects performance and reliability. To achieve interoperability quality attribute scenarios like I2, a REST service may need to use different representations of resources suitable for different devices and systems.

2) Is a standardized vocabulary defined for the resource?

A resource representation should follow a predefined vocabulary. For XML documents, the vocabulary can be defined by an XML schema. The vocabulary should also be standardized across all services that handle that same resource, as previously discussed in question A.3. Using a vocabulary known by service consumers improves interoperability scenario I3. Also, all information needed to understand the resource must be included in the request and response messages (constraint Stateless, session IV). The HATEOAS (Hypermedia as the Engine of Application State) constraint [3] is used for this purpose. This constraint states that the hypermedia document that represents the resource should be used to find associated resources. For instance, the following fragment of the representation of resource "course" from a university contains a link to another resource representing the lecturers that teach the course:

```
<course>
  <name> Software Architecture </name>
  <area>Software Engineering</area>
  <instructors>
     <uri>/courses/sa/lecturers</uri>
  </instructors>
</course>
```

HATEOAS positively affects discoverability scenario D1. The response time for a service request decreases because the representation does not need to bring all information about the linked resources, but only references (hyperlinks). However, a greater number of network requests may be necessary to retrieve linked information and the overall performance may be negatively impacted.

3) How do you design resources URIs?

The Uniform Resource Identifier (URI) is a string of characters used to uniquely identify a web resource. A resource must have at least one URI that makes it addressable via http. Several principles for designing URIs [11] can be used for the specification of REST resource identifiers.

Designing descriptive URIs related to the hierarchy of the domain model improves reliability scenario R2. Examples of descriptive URIs are listed below:

- http://www.myweather.com/current/city/Brasilia
- http://www.ufrj.edu/courses/programming101

However, URIs are not always descriptive. An example of URI where a resource is identified by a numeric ID (52545) is http://www.resoucerslib.com/52545. Resource identification by ID positively impacts the modifiability quality attribute scenario M2, but impacts negatively in reliability scenario R2. Other strategies to design resource identifiers can be used. Evaluators should inspect the trade-offs between descriptive and non-descriptive URIs according to both the usability and adaptability requirements. Establishing a pattern to define URIs for REST services is a good practice that positively impacts the functionality scenario F1.

In general, URIs should use nouns, not verbs. According to the Uniform Interface constraint, the operation to be executed should be specified by the http verb, not in the URI. For example, instead of URI http://www.ufrj.com/getcourses, we should use http get on URI http://www.ufrj.com/courses. If verbs are used in the URI, functionality, as in scenario F5, could be negatively impacted.

4) What is the approach for resource versioning?

Representations of resources are accessed by their identification (URI). If a representation is modified, service consumers that request that resource can be negatively impacted. The reason for such modification can be evolution of the domain model that impacts in the resource representation. Versioning of resources is a good practice to solve this problem that positively affects the modifiability scenario M3. A commonly used strategy to version resources is to include a version number within the URL. For instance:

- http://www.ufrj.com/courses/v1/comperscience

Another alternative is to include the version in the http header, for example: "Accept: *Content-type: application/xml;* ***version=1.0***". Evaluators should inspect the trade-off between URI versioning and http header versioning. For example, for human users requesting resources via web browsers, URI versioning may be a better solution and, in this case, reliability quality attribute scenario R1 is positively impacted.

5) How do you map operations on resources to http verbs?

This question is related to question 2 (*How to Identify Resources?)*. Per the Uniform Interface constraint (Section IV), the interface must use common http methods (also called verbs) to indicate the action to be performed on a resource: POST to create a new resource; PUT to update a resource; DELETE to remove a resource; and OPTIONS to list what methods are supported on a resource. By using http verbs functionality scenarios like F5 are promoted.

Important concepts when designing the resource exposure are idempotent and safe methods. Safe methods are http methods that do not modify resources (for instance GET and OPTIONS); they only request information. An idempotent http method is a method that can be called many times without different outcomes. For instance, consider two actions that assign a value to variable *var*: (a) var = 5; (b) var++. The first example (a) is idempotent; no matter how many times the method is executed the result will always be 5. The second example (b) is not idempotent. Different number of execution results in different outcomes. Http PUT and DELETE are idempotent, POST is not. Safety quality attribute scenarios Sa1 and Sa2 are related to idempotent and safe methods.

C. Documentation and testing

Service consumers perform actions (GET, PUT, POST and DELETE) in resources. Evaluators should inspect the documentation about resources related to identification, representation, security and design. Developers of service consumers need to understand the service interface. When evaluating documentation and tests, the following questions help determining risks:

1) How resources should be documented?

Developers of service consumers need to understand how to interact with the service interface to access and use resources. Effective service interface documentation positively impacts the functionality scenario F2.

2) How service consumers can perform tests in resources?

It is important for service consumers and developers to perform tests in the service interface. Dynamic tests can be executed using a "service sandbox". Http methods like GET and PUT can be executed against the sandbox environment. This is a typical scenario to improve the testability quality attribute scenario T1.

D. Behaviour

Behaviour is related to the service interface behaviour when service consumers perform actions on resources. When evaluating behaviour, the following questions help determining risks:

1) What are the http status codes in responses?

REST services should use standard http status codes to report success and error when processing requests. Evaluators should ask about the use of standard http status codes. Their use contribute to functionality scenarios, such as F5.

2) Does the response http header contain information about the resource?

When service consumers perform actions in a resource, the service implementation must include in the http responses the header field "Location" filled with the URI of the manipulated resource. Service consumers can correlate the URI of the new resource created with the URI in the Location header response, positively affecting functionality scenarios like F3.

3) Can the resource communicate with Open APIs?

Open API is a common trend for web-based services in social networks and e-business, to publish public services over the Internet. If a service interface allows the use of Open APIs, functionality (for instance scenario F5) is positively impacted. However, the trade-off between functionality and security (as in scenario S1) should be considered.

4) Is pagination in resources necessary?

Full representation of resources may negatively impact performance scenarios like P1. Evaluators should assess if pagination of the resource representation is needed. A good practice is to use query strings to specify page number and size. For example, a URI for "courses at UFRJ" that specifies the visualization for page one with thirty courses could be: http://www.ufrj.com/courses?page=1&size=30. Pagination is related with performance and functionality scenarios, such as P1 and F4.

5) Is it possible to include the subset of desired attributes in the URI?

Suppose a service consumer only wants a subset of attributes and the service interface only supports responses containing the full representation of a resource. In this case, performance and functionality scenarios P1 and F4 are negatively impacted. A good practice is to include a filter in the URI. For example, the following URI will retrieve only the name and number of credits for courses at UFRJ: http://www.ufrj.com/courses?attributes= name,credits.

6) How to protect the web server from request overload?

Service consumers can perform a huge number of requests resulting in web server overload. Evaluators should ask about mechanisms in place to prevent request overload, and hence promote promote availability, as in scenario Av1.

E. Security

Design questions about security are related to authentication, authorization, and privacy. When evaluating security, the following questions help determining risks:

1) Did service design consider information classification?

This question is associated to design question 2 (*What data will be exposed as resources?*). Evaluators should pay special attention to the risk of exposing private information in public resources. Different parts of a resource may have different security classifications. Such situation may impact granularity of the services (e.g., breaking up the resource into two, one with public information and the other with private information). Scenarios such as S1 are affected.

2) What are the security mechanisms for service consumers to perform actions on resources?

Some resources are restricted to specific groups of service consumers. Security mechanism to ensure authentication and authorization of service consumers may be required. Some

commonly adopted standards are OAuth [12] and OpenID [13]. Http Basic Authentication can also be used. The version 2.0 of OAuth has proven to be a good strategy for secure authorization in REST-based architectures. A typical scenario for related to this question is S2.

VII. Evaluation Proof of Concept

The goal of this section is to show the use of the proposed guidance in a proof of concept system evaluation. The scenario-based method employed is the Architecture Trade-off Analysis Method (ATAM) [5] and it was applied to a Web-based system, briefly described as follows.

A. Architecture Trade-off Analysis Method (ATAM)

The ATAM method provides a principled way to evaluate the fitness of a software architecture with respect to multiple competing quality attribute requirements. The method was created to uncover the risks and trade-offs reflected in architectural decisions related to quality attribute requirements. The ATAM method consists of several activities (as described in Section II). The documentation about the method, with all ATAM steps, can be found at http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm.

The general scenarios proposed in this paper can be used in different activities of ATAM. Our proof of concept focused on the activities: 3 (Present architecture), which is described in subsection C; 4 (Identify architectural approaches), which is descripted in subsection D; concrete scenarios of activity 5 (Generate quality attribute utility tree), described in subsection E, and on the use of guidance in light of activity 6 (Analyze architectural approaches), which is described in subsection F.

B. System description

The system used to evaluate the guidelines proposed in this paper was EcoDiF (*Web Ecosystem of Physical Devices*) [14]. Interoperability is an important quality attribute requirement for EcoDiF, so the architecture was designed according to REST principles and constraints. EcoDiF is available at http://ubicomp.nce.ufrj.br/ecodif/index_en.html. It is a Web platform to connect devices and products with applications and/or end users in order to provide control, visualization, data processing and storage functionalities. Fig. 3 shows an overview of EcoDiF, illustrating the integration of heterogeneous physical devices and the use of data provided by different kinds of users.

EcoDiF has four types of stakeholders: (i) *devices manufacturers*, which develop drivers to their devices to make them compliant with the EcoDiF open API, as well as data profiles that specify the metadata that describes the type of data provided by the their devices; (ii) *data providers*, which are device owners that want to make the data produced by their devices available at the IoT ecosystem through EcoDiF; (iii) *application developers*, which build Web applications or services that use as inputs the data available at EcoDiF as well data produced by any other Web accessible resource, and; (iv) *information consumers*, which are users that interact with the EcoDiF ecosystem.

C. EcoDiF Architecture Overview

Fig. 3 also shows the main modules in EcoDiF. The *Devices Connection module* aims to facilitate the connection of physical devices to EcoDiF and, consequently, to the Internet. Manufacturers configure their devices according to the EcoDiF interface to enable the integration with the platform. Users connect their preconfigured devices to perform the operations of the provided interface. This connection between the device and EcoDiF is enabled by a customized driver to the specific device, so that data providers can use such driver for connecting their devices and make the data obtained from the devices available at EcoDIF. Data messages (called feeds) are represented using EEML (*Extended Environments Markup Language*) [15], an XML-based language that describes data obtained from devices in a specific context. The data is sent by the device to EcoDiF through a driver through http PUT, and users can manipulate the data through EcoDiF *Data Manipulation Component*.

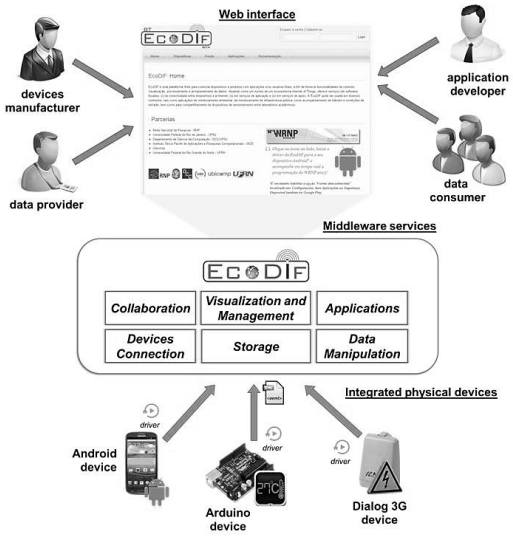

Fig. 3 EcoDiF Architecture

The *Visualization and Management module* provides a Web interface for users managing the devices connected to EcoDiF. These users can monitor the state and location of their devices, create alerts and notifications (*triggers*) about the environment, and visualize historical data. The *Collaboration module* let EcoDiF users perform searches for devices and applications registered with the platform based on their metadata (type, user, location, etc.). The *Storage module* consists of two repositories, a relational database for the data feeds, and a repository for *storing application* scripts in a file system. These repositories may use a Cloud Computing infrastructure to store relational data and files to enhance the reliability, security, availability, and scalability of the solution.

Finally, the *Applications module* provides a model and environment for programming and executing applications that can use the data (feeds) available at EcoDiF and generate new information to be available at the platform. In EcoDiF, these applications are built as Web *mashups* [16]. The EMML (*Enterprise Mashup Markup Language*) [17] XML-based language is used to develop Web mashup applications by

integrating data from several sources, including Web services, third-party APIs, and relational databases.

D. Architectural Approaches of EcoDiF

The Layered System (*L*) constraint is applied by grouping responsibilities in three tiers (Fig. 3): Web Interface, Middleware Services, and Integrated Physical Devices. EcoDiF acts as a web server and physical devices act as Clients (see Section IV). The Stateless (*S*) constraint is applied by not keeping state of requests; all resource information is included in the response. It does not have HATEOAS. Code on Demand (*CoD*) and Cache (*$*) constraints are not applied. Resources were designed according to the EEML protocol. The only resource exposed to clients is a feed. When a user provider creates a feed, EcoDiF assigns an ID to uniquely identify the feed. The URI pattern for identifying the feed is: ecodif.com / api / feed / [ID]. The http method used to request a feed is GET. The same URI is used by user providers to update feeds using http PUT. When a feed is updated, EcoDiF responds with http status code 201, but does not include the URI of the updated feed. To perform a feed update by using PUT, the user credentials must be added as an http header. User providers get their credentials upon registration. If the user provider configures the feed as public, GET is performed without security validations. If the feed is private, only the creator can request the feed representation. The feed uses the application/xml media type. When a resource is requested, EcoDiF sends the full representation of a feed to the client. EcoDiF does not have versioning of resource URI. The EcoDiF interface documentation is available on the web.

E. Concrete Scenarios

Table II shows three quality attribute scenarios for EcoDiF that are relevant to a REST-based architecture. The complete analysis of EcoDiF's architecture, including all elicited concrete scenarios, can be found at: http://ubicomp.nce.ufrj.br/restguidance/

TABLE II. CONCRETE SCENARIOS FOR THE ECODIF

Quality Attribute	Concrete scenario
Scenario 1. Interoperability	**(Source)** EcoDif's driver/data provider **(Stimulus)** Provider sends data to EcoDiF platform **(Artefact)** Driver/EcoDiF's interface **(Environment)** Normal operation **(Response)** Sensor data is sent and Driver receives the confirmation from EcoDiF's interface **(Response Measure)** The system responds to the driver in less than five seconds
Scenario 2. Interoperability	**(Source)** Data consumer **(Stimulus)** Data consumer requests feed data **(Artefact)** EcoDiF's interface **(Environment)** Normal operation **(Response)** The system sends to the user the resource representation of feed **(Response Measure)** The system responds to the driver in less than 1 second with the correct requested media type
Scenario 3. Testability	**(Source)** Devices manufacture **(Stimulus)** Devices manufacture wants to create a new driver and perform tests in EcoDiF's API **(Artefact)** Devices connection **(Environment)** Normal operation **(Response)** The system offers documentation for driver's development **(Response Measure)** The documentation offers all information to develop and test the driver

F. Architectural Analysis

The architecture approaches were probed by using guidelines referenced in Section VI for each scenario.

TABLE III. ANALYSIS FOR CONCRETE SCENARIO 1

Scenario Summary	Sensor data is sent to EcoDiF by the driver. System updates the feed and responds to the Driver in less than five seconds
Business Goal(s)	Allow communication of heterogeneous devices with EcoDiF
Quality Attribute	Interoperability, Availability, Performance, Functionality
Architectural Analysis	-Domain-based vocabulary (EEML) promotes Interoperability. (**see subsection B - Representation and identification, question 2)** - By using only XML media type interoperability is negatively impacted (**see subsection B - Representation and identification, question 1)** - The URI is designed by feed code. It promotes extensibility but negatively impacts in interoperability (**see subsection B - Representation and identification, question 3**) - The method used to update resources is PUT and it is idempotent. It promotes Reliability. (**see subsection B - Representation and identification, question 5**) - EcoDiF's interface responds with http code 201 but does not include the feed URI in response. (**See subsection D - Behaviour, question 2**)
Risks	- Drivers that do not use XML could not send information to EcoDiF - The design does not respond with the resource URI in header Location
Trade-offs	- The use of different media types promotes interoperability but adds complexity to implementation and validation.

TABLE IV. ANALYSIS FOR SCENARIO 2

Scenario Summary	Consumer requests information about some feed and receives it in less than 1 second with the correct media type requested.
Business Goal(s)	Allow easy visualization of feeds
Quality Attribute	Interoperability, Availability, Functionality, Adaptability
Architectural Analysis	- The URI is designed by the feed code. It promotes extensibility but negatively impacts usability (**see subsection B - Representation and identification, question 3**) - It is not possible to explicit the media type in requests. Interoperability is negatively impacted - It is not possible to explicit the version of requested resources. Adaptability is negatively impacted (**see subsection B - Representation and identification, question 4**) - It is not possible to include requested attributes in the URI. Performance and usability is negatively impacted (**see subsection D - Behaviour, question 5**) - Resources are not linked to others. Navigability is negatively impacted (**See subsection B - Representation and identification, question 2**)
Risks	- The design does not meet the requirement in this scenario. The system can only produces XML

	responses; it is not possible to specify neither the version nor desired fields in the resource URI; HATEOAS is not used, thus the service consumers cannot navigate to other associated resources.
Trade-offs	- Adaptability is promoted when using versioning of resources, but it impacts in the configuration management of resources - Expliciting fields in request improve performance, but adds complexity to implement filtering in the service interface

TABLE V. ANALYSIS FOR SCENARIO 3

Scenario Summary	Devices manufacture needs to create a new driver and EcoDiF offers all information necessary to develop and test the driver.
Business Goal(s)	Facilitates the development of Drivers
Quality Attribute	Usability, Interoperability, Testability
Architectural Analysis	-EcoDiF's interface documentation offers information in text documents. (**see section C - Documentation and testing, question 1**) - It does not have a sandbox for testing the drivers (**see section C - Documentation and testing, question 2**)
Risks	-Without a sandbox for testing, device manufacturers cannot analyse the response behaviour or error messages before deliver the driver for data providers.
Trade-offs	-Developing a sandbox is important to improve development and testing of new Drivers. However, the sandbox approach negatively impacts modifiability since modifications in service interface must be done in the sandbox as well.

VIII. DISCUSSION AND FUTURE WORK

The main contribution of the paper is the collection of general quality attribute scenarios and design questions that can assist evaluators in REST-based architecture evaluations, improving the mitigation of risks. As shown in Section VII the guidelines can be used in scenario-based methods such as ATAM to help evaluators to probe architectural approaches in order to identify trade-offs and risks to addressing quality attribute requirements.

The research protocol proposed in this paper has a limitation related to the validation of design questions: it does not define an activity to evaluate the design questions that were elicited in interviews and literature review. So, future work should include validation activities in the protocol. Furthermore, it is important to investigate other elements that influence design questions and quality attribute requirements for REST systems, such as: *(i)* legacy systems integration; *(ii)* metadata representation; and *(iii)* security trade-offs with SSL. We also intend to investigate how Cloud Computing features impact REST design questions and general quality attribute scenarios.

Acknowledgment

This work was partially supported by Brazilian Funding Agencies FAPERJ (under grant E-26/102.961/2012 for Flavia C. Delicato), CNPq (undergrant 311363/2011-3 for Flavia Delicato and 310661/2012-9 for Paulo F. Pires), COPPETEC/UFRJ, and UFRJ/CENPES. We thank to all specialists that contribute to research in special to Matthias Naab (Fraunhofer IESE); Daniel Jacobson (Netflix), Casare Pautasso (Università della Svizzera Italiana); Erik Wilde (UC Berkeley School of Information); Felipe Oliveira and Alexandre Saudate (SOA Expert); Luiz Cipriani and Luiz Rocha (Abril Midia); Kleber Bacili (Sensedia); Wanderlei Souza (Apontador.com), engineers from Google Maps API, and Architecture Evaluators from SEI.

References

[1] Bass L., Clements P., Kazman R., Software Architecture in Practice (SEI Series in Software Engineering). 3rd ed. Addison-Wesley Professional: 2012.

[2] Naab M., "All Architecture Evaluation is not the Same – Lessons Learned from more than 50 Architecture Evaluations in Industry," in Architecture Technology User Network (SATURN), 2013

[3] Bianco P., Kotermanski R, Merson P., "Evaluating a Service-Oriented Architecture". Software Engineering Institute (SEI), Technical Report. CMU/SEI-2007-TR-015, September, 2007.

[4] Fielding T. F., "Architectural Styles and the Design of Network-based Software Architectures". Ph.D. dissertation, Univ. of California, Irvine, 2000.

[5] Kazman R., Klein M., Barbacci T., Longstaff T., Lipson H., "The Architecture Tradeoff Analysis Method". In proceedings ICECCS, pp. 68-78, 1998

[6] Oishi S. M., "How to Conduct In-Person Interviews for Surveys", 2nd ed. ThousandOaks, CA: Sage Publications, 2002

[7] Kasunic M., "Designing an Effective Survey". Software Engineering Institute (SEI), Handbook. CMU/SEI-2005-HB-004, 2005

[8] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. Documenting Software Architectures: Views and Beyond, Second Edition. Addison-Wesley, 2010

[9] Richardson L., Ruby S., "RESTful Web Services", 1st ed., O'Reilly Media, 2007

[10] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T. " Network Working Group Request for Comments: 2616". Internet Engineering Task Force: 1999. [Online] Available: http://www.ietf.org/rfc/rfc2616.txt

[11] Berners-Lee T., Universal Resource Identifiers -- Axioms of Web Architecture [Online]. Available: http://www.w3.org/DesignIssues/Axioms.html

[12] OAuth Community Site [Online]. Available at: http://oauth.net/

[13] OpenID Foundation website [Online]. Available at: http://openid.net/

[14] Delicato F. C., Pires P. F, Batista T., Cavalcante E., Costa B., Barros T., "Towards an IoT Ecosystem". In SESoS, pp. 25-28, 2013.

[15] Extended Environments Markup Language [Online]. Available at http://www.eeml.org/.

[16] Guinard, D.; Trifa, V.; Pham, T.; Liechti, O., "Towards physical mashups in the Web of Things," INSS 2009, vol., no., pp.1,4, 17-19 June 2009.

[17] OMA EMML Documentation [Online]. Available at:http://www.openmashup.org/omadocs/v1.0/index.http

[18] Erl T., Carlyle B., Pautasso C., Balasubramanian R., Wilhelmsen H., and Booth D., "SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST" (The Prentice Hall Service Technology Series from Thomas Erl). Prentice Hall, 2012

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

[20] Vinoski S., "REST Eye for the SOA Guy," IEEE Internet Comput., vol. 11, no. 1, pp. 82–84, Jan. 2007.

[21] Wilde E. and Pautasso C., "REST: From Research to Practice". Springer, 2011.

[22] Barbacci, M., Klein, M., Longstaff, T., Weinstock, C., "Quality Attributes". Software Engineering Institute (SEI), Technical Report. CMU/SEI-95-TR-021, December, 1995.

[23] International Organization for Standardization. "System and software quality models". ISO/IEC 25010:2011.