The Fallacy of the Multi-API Culture: Conceptual and Practical Benefits of Representational State Transfer (REST)

Ruben Verborgh[1]*, Seth van Hooland[†]
Aaron Straup Cope[‡], Sebastian Chan[‡],
Erik Mannens[1], and Rik Van de Walle[1]

[1] Ghent University – iMinds – Multimedia Lab
Gaston Crommenlaan 8 bus 201
B-9050 Ledeberg-Ghent, Belgium
{ruben.verborgh,erik.mannens,rik.vandewalle}@ugent.be

[2] Université Libre de Bruxelles
Information and Communication Science Department
Avenue F. D. Roosevelt, 50 CP 123
B-1050 Brussels, Belgium
{svhoolan,madewild}@ulb.ac.be

[3] Cooper-Hewitt National Design Museum
2 East 91st Street
New York, NY, 10128, USA
{CopeA,ChanS}@si.edu

Abstract

Purpose The paper revisits a decade after its conception the Representational State Transfer (REST) architectural style and analyses its relevance to address current challenges from the Library and Information Science (LIS) discipline.

Design/methodology/approach Conceptual aspects of REST are reviewed and a generic architecture to support REST is presented. The relevance of the architecture is demonstrated with the help of a case-study based on the collection registration database of the Cooper-Hewitt National Design Museum.

Findings We argue that the "resources and representations" model of REST is a sustainable way for the management of Web resources in a context of constant technological evolutions.

Practical implications When making information resources available on the Web, a resource-oriented publishing model can avoid the costs associated with the creation of multiple interfaces.

Originality/value This paper re-examines the conceptual merits of REST and translates the architecture into actionable recommendations for the LIS discipline.

*Corresponding author

1 Introduction

1.1 General context

"We shall be questioning concerning technology, and in so doing we should like to prepare a free relationship to it". With this agenda in mind, Heidegger opens his 1954 essay on the essence of technology and humanity's deeply-entangled relation with it (Heidegger, 1954). Interpretations of this essay notoriously vary, but his key message is to raise awareness regarding our inability to step outside technological thinking. With this message, Heidegger is often mistaken for a Luddite, but there is tremendous value in the quest to see technology for what it really represents. By doing so, we can try to manage its impact on our lives.

Much of the same thinking underpins the introduction of Roy T. Fielding's doctoral thesis (Fielding, 2000), which formalized the concept of the Representational State Transfer (REST) architectural style for distributed hypermedia systems such as the Web. Fielding criticizes in his introduction the "design-by-buzzword" context in which Web applications are developed. In a distributed network such as the Internet, innovation through fast-paced technological changes comes at a cost. The expense of *adaptation* considerably drives up the costs of our information industry.

More than a decade later, the desire for new applications continues to grow, particularly for rich Web applications and mobile applications (Kroski, 2008). The re-use of Web content in these contexts requires an automated access to the content in a more rigidly structured format than HTML, such as for example JSON (Severance, 2012). If the incentives are important enough, the owner of the Web content typically invests in developing a *Web API* that replies in the JSON format. A Web API or Web Application Programming Interface is a set of protocol constructs offered by a Web application through which third-party Web or software applications can interact with it. It typically has a large functional overlap with the possibilities offered to visitors offered by the website, but a Web API makes those available to pieces of software as well, such as JavaScript components on an internal or external Web site, mobile applications, desktop applications, and others. A related concept is *Web service*, which predates the term "Web API", and refers to a means to enable programming over the Web. In practice, definitions notoriously vary; however, "Web services" have come to stand for heavy-weight, enterprise, XML-messaging-based solutions, whereas "Web APIs" are more light-weight and targeted at Web and mobile applications.

Interestingly, a Web API often does not offer new functionality in addition to the HTML version; it mostly provides the same content and actions, but makes them available in another format. The logical consequence of building new APIs in response to changing requirements is that a separate API will be necessary for each and every purpose. If, for example, a website wants to make content available in RDF, it would also need an RDF API next to the existing HTML and JSON APIs. Additionally, to meet social network demands, it might also need special APIs, and maybe even others for specific device types. In the end, the multiplication of APIs results in a website's content being scattered across applications which all need to be maintained separately.

URLs (Uniform Resource Locators) can serve as a window to observe the fast-evolving and hype-driven environment. The multi-API culture has manifested itself over the years in the appearance of implementational details within URLs. For instance, imagine a URL containing `/showObject.php?id=3685`. The presence of `showObject` points to a specific script and `php` indicates the use of a specific technology. In the short term, these characteristics are not *necessarily* problematic as the URL fulfills the function of uniquely identifying a resource. However, if we allow the underlying technology to influence the naming scheme, each change in technology can possibly change the URL which serves to identify a resource. This concrete example is a painful illustration of the need to *decouple* as much as possible the identification of resources and the concrete technologies used to manage them.

1.2 Relevance to the LIS discipline and research question

Even though REST has been primarily discussed within the Computer Science domain (Khare and Taylor, 2004; Pautasso *et al.*, 2008; Vinoski, 2007; Zuzak and Schreier, 2012), its implications for the LIS discipline are evident and have already been discussed in a number of papers. The earliest reference discusses the evolution of the JISC

Information Environment from a SOAP-based approach towards RESTful services to facilitate the integration of heterogeneous information resources within a common virtual infrastructure (Powell, 2005). (Guy, 2009) documents, also within the JISC context, the benefits and challenges of APIs for the dissemination of information resources. In the wake of the Linked Open Data (LOD) movement, a number of publications addressed the use of REST for the publication of controlled vocabularies on the Web. Most notably, Panzer (2008) describes the challenges involved for the online publication of the Dewey Decimal Classification and the role of REST. Within the same conference proceedings, Summers *et al.* (2008) detail their work on the publication of LCSH as Linked Data. Within a more recent working paper, Summers gives an overview of best practices for the publication of Linked Data for libraries, archives and museums, in which REST is also discussed (Summers and Salo, 2013).

Although REST has been considered in the existing LIS literature, no previous publication has provided a thorough introduction on the topic. This paper specifically examines the relevance of REST more than a decade after its development. We will investigate what conceptual and practical benefits REST offers for collection holders to develop a future-oriented, forwards-compatible Web architecture. REST potentially has a mayor role to play, as it focuses attention on *"scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems"* (Fielding, 2000).

In particular, we will assess the relevance of REST in connection with the growing popularity of LOD. Recent initiatives such as OpenGLAM[1] and LOD-LAM[2] illustrate how these evolutions are percolating into the cultural heritage domain. Both the US and the EU flagship digital library projects, respectively the Digital Public Library of America[3] and Europeana[4], are currently embracing Linked Data principles (van Hooland *et al.*, 2012). As the above example with the URL containing `/showObject.php?id=3685` demonstrated, the culture of multiple APIs holds a danger for the longevity of URLs, undermining the successful implementation of Linked Data principles. This paper will therefore analyze what role REST can play within the development of an information architecture capable of issuing and maintaining persistent identifiers for the objects an institution manages.

1.3 Methodology

In order to side step a purely conceptual discussion regarding the benefits of the model, the article introduces a case-study in which the REST principles are implemented. The confrontation between the model and the empirical reality of the use case avoids the risk of presenting a merely dogmatic stance regarding the value of REST principles. While doing so, the paper also wants to keep away from the "black box problem". Mentioned by Ramsey and Rockwell in the context of the Digital Humanities domain, the term refers to the difficulty of adequately describing the value of the implementation of a technology. When presenting research, *"[...] either the technique is encapsulated inside the black box of magical technology or it is unfolded in tedious detail obscuring the interpretation – tedious detail which ends up being a black box of tedium anyway"* (Ramsay and Rockwell, 2012).

To bypass this problem, the paper has very consciously tried to find within the presentation of the use case an optimal balance between the provision of sufficient practical, technical and conceptual background and details. Through a combined practical and technical understanding, researchers and practitioners from the LIS discipline can evaluate the conceptual added value of REST principles.

Within the context of this paper, we chose the cultural heritage sector as an application domain. Libraries, archives and museums have been very active over the last decade to make their resources available on the Web. As other sectors, these institutions have suffered from the fast-evolving technological landscape (Boydens and van Hooland, 2011). The recent interest in Linked Data within this community described in Section 1.2 stresses the importance of having a robust information architecture. Within the cultural heritage sector, a choice was made to

[1] `http://openglam.org`, accessed July 20, 2013
[2] `http://lodlam.net`, accessed July 20, 2013
[3] `http://dp.la`, accessed July 20, 2013
[4] `http://europeana.eu`, accessed July 20, 2013

present a paradigmatic use case, which aims *"to develop a metaphor or establish a school for that domain that the case concerns"* (Flyvbjerg, 2006). The Smithsonian Cooper-Hewitt National Design Museum is very actively reflecting upon the use of the Web to make its resources as easily available as possible. In many regards, the museum is one of the most interesting international players in regards to collection dissemination on the Web, making the Cooper Hewitt a relevant choice as a case-study. More details regarding the use case and to what extent the results are generalizable to other institutions are presented in Section 5.

1.4 Outline of the paper

To understand the obstacles that have made active use of this architecture difficult until today, a brief overview of online information architecture from the Web's beginning until now is given in the next section. In Section 3, we will provide a detailed explanation of the conceptual model for long-term sustainability. Section 4 introduces a generic architecture to implement this model, discussing the impact of change on systems that use this architecture. The proposed architecture is then illustrated with the help of a real-life use case (Cooper-Hewitt Museum) in Section 5, helping the reader to understand the added-value of REST for the documentation of cultural heritage collections. We conclude the article in Section 6.

2 Short history of Web architecture

2.1 Relevance of original HTTP design

In the early days of the Web, only documents written in HyperText Markup Language (HTML) could be distributed because the communication mechanism at the time, version 0.9 of the Hypertext Transfer Protocol (HTTP), solely supported HTML (Berners-Lee, 1991). In 1993, Mosaic was the first browser to support embedding images in hypertext documents, thereby starting the multimedia era on the World Wide Web. By the time HTML 2 arrived, implementations of HTTP were able to deal with different types of content. The HTTP 1.1 specification, released in 1999, offered forward-compatible support for all possible content types and detailed central concepts such as connections, messages, methods to view and manipulate information, authentication, and caching (Fielding *et al.*, 1999).

Although the underlying transfer protocol supported arbitrary content types, native browser support was only provided for simple types such as text, HTML, and basic image formats (GIF and JPEG). The scripting language JavaScript could offer a higher degree of interactivity, but its presentational possibilities remained confined to those of HTML. Therefore, several plugins (notably Flash Player and Java) were developed that basically could render and manage a specific content area of HTML pages (Wilde, 1999). HTTP always remained the transport protocol of data.

Below is an example HTTP request, sent by a client to a server. In the top line, we notice a method (`GET`), the relative URL of the requested resource (`/objects/35460799/`), and the HTTP version (`1.1`). Below this initial line, we see several key/value pairs separated by a colon, attaching metadata to the request. For instance, we see the user's browser (`User-Agent`), accepted document types (`Accept`), and language preferences (`Accept-Language`).

```
GET /objects/35460799/ HTTP/1.1
Host: collection.cooperhewitt.org
Connection: keep-alive
Cache-Control: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Pragma: no-cache
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/537.36
            (KHTML, like Gecko) Chrome/28.0.1500.71 Safari/537.36
Accept-Language: en-US,en;q=0.8
```

To this, the server replies with an HTTP message that includes a status code `200`, indicating success. The reply also contains metadata fields, for instance, the type of the reply body (`text/html`). At the end of the message, the requested document is added.

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Length: 6796
Content-Type: text/html; charset=utf-8

<html>
  ...
```

This is the essence of HTTP communication: a client sends a request message, to which the server replies with an answer message that contains the document.

2.2 Unique identifiers and locators for resources

The previous section indicates the importance of identification and location on the Web. The difference between these two aspects can be illustrated by identifiers that perform only one of two functions. For instance, a social security number uniquely *identifies* a person: given this number, there will be at most one individual who corresponds to it. However, a social security number does not provide the means to locate a person directly. In contrast, a home address allows to *locate* a person, but because several people might live at any given address, this address alone is insufficient to identify a single person. Furthermore, addresses can change over time, whereas a social security number does not.

The Web's most well-known identification mechanism, the Uniform Resource Locator (URL), was specifically designed to support identification and location at the same time. A URL can indeed serve as an identifier for something on the Web, as is the case with `http//collection.cooperhewitt.org/objects/35460799/`. Additionally, the URL will detail how to retrieve the resource it identifies because of its special structure. To retrieve a representation of the collection object, we initiate a connection to the host (`collection.cooperhewitt.org`) and then issue an HTTP request with the remainder of the URL (`/objects/35460799/`), as illustrated above.

Uniform Resource Identifiers (URIs) are a generalization of the URL concept. They only allow unique identification, but not always location. Very broadly speaking, any distinguishable concept in the universe can be assigned a URI. URLs are a particular kind of URIs, but infinitely many others exist. For instance, ISBN numbers can be written as URIs, with the ISBN number 978-0062515872 corresponding to `urn:isbn:978-0062515872`. Note how this number indeed identifies a book, but does not directly allow to retrieve a copy of it.

More recently, the concept of URI has been generalized to IRI, the Internationalized Resource Identifier. URIs are limited within the ASCII character set, whereas IRIs allow the inclusion of Unicode characters.

2.3 Appearance of Web services and multiple APIs

Around 2005, the diversity of the browser landscape started increasing. New browsers emerged, and more and more devices started joining the Web: personal digital assistants, smartphones, and eventually tablets. Websites that previously only targeted the average computer with a *de facto* standard configuration, suddenly needed to become compatible with various browsers and screen sizes. More complicated applications were developed that make use of more and more Web APIs.

The response to this change has been to add more interfaces to the existing infrastructure: an API was created for each type of client. This multi-API approach can be seen in Figure 1, which clearly illustrates the substantial overhead in development and maintenance costs. At first, each API has to be designed and implemented separately. Next, if new functionality is required, each API has to be updated. Furthermore, it is difficult for different types of clients to communicate to each other, as they all interact with the database in an different way. Finally, if a back-end

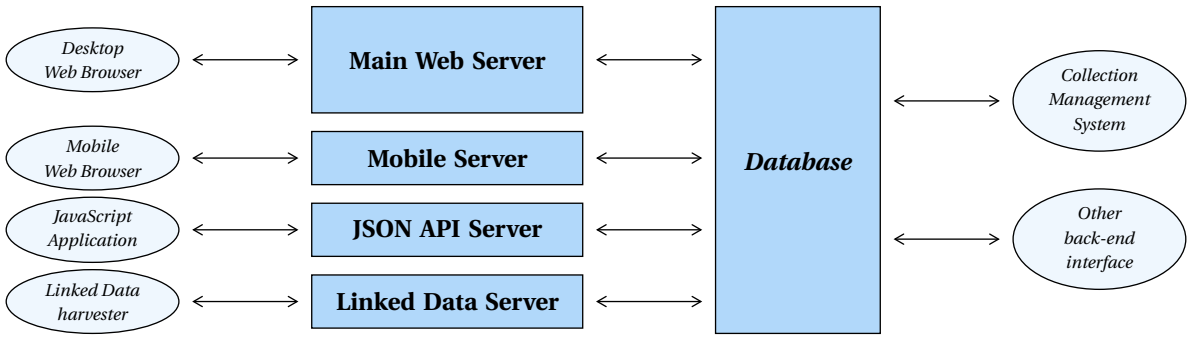

Figure 1: The costly model of the multi-API: new interfaces must be built and maintained for each type of client (desktop / mobile / application / harvester / ...) that accesses the collection. Furthermore, as each interface directly depends on the database, changes enforced by the internal Collection Management System will have to be dealt with in all interfaces separately.

system requires changes to the database, all interfaces—and possibly the client-side code to those interfaces—have to be updated. This multi-API might have been the right approach to respond to short-term change, but it is no longer sustainable on a Web in constant change.

2.4 Role of Linked Data and need for uniquely addressable resources

From the year 2000 onwards, it became clear that humans were not going to be the only consumers of the Web. Software developers wanted to create tools that could use the Web as a source of information and as a platform to perform actions. Web services were created using technologies based on the Extensible Markup Language (XML), and they allowed software clients to act on them automatically. There was, however, a clear barrier between the resources intended for humans (HTML) and those for machines (XML) (Verborgh *et al.*, 2013). Today, services are called "Web APIs" as they are migrating to the JavaScript Object Notation (JSON) format, which is easier to be handled by JavaScript applications.

While Web APIs mostly focus on performing *tasks* over the Web (adding, modifying, or annotating information) Linked Data principles focus on making *data* machine-accessible (Bizer *et al.*, 2009). Although HTML can define the structure of document data, it does not provide information about its content. Therefore, Linked Data is expressed in the Resource Description Framework (RDF) data model, allowing software applications to interpret this content and transform it in numerous ways. For this model to be successful, we need persistent identifiers under the form of URLs. The persistence of URLs is necessary to guarantee that we can build new datasets upon them, ensuring that the identifiers remain valid. Other reasons for persistent URLs are discussed by Gomes and Silva (2006) and McCown *et al.* (2005).

3 The REST model and its implications

3.1 Definition

Rather than a specific technology or standard, REST is an architectural style that *"has been used to guide the design and development of the architecture for the modern Web"* (Fielding, 2000). It is important to note that REST is a *style* for system architectures, dictating several *architectural constraints*, rather than a technology or architecture in itself. The most widespread architecture that is subject to these constraints is of course the World Wide Web itself, whose transfer protocol HTTP is governed by the REST principles.

However, this does *not* imply that every website therefore conforms to the REST constraints *by default*. On the contrary, many Web application frameworks enforce a style that is not compliant with the REST architectural principles. Similar to the situation where an urban architect designs a neighborhood according to several structural

constraints, individual websites can (and do) disregard the governing principles, just like individual office buildings in the complex can be designed by different architects. Although not the desired overall solution, it occurs frequently on the Web, where it is not a question of aesthetics but a matter of integration and scalability. In this section, we will investigate two groups of REST constraints: the *client-server constraints*, and the *uniform interface constraints*, which are highly specific to this architectural style.

3.2 Client-server constraints

The first set of constraints make REST architectures client-server systems, where a *client* sends requests to a *server* that offers a service the client is not able (or willing) to perform itself (Sinha, 1992). On the Web, the principle actors in these roles are Web browsers and Web servers. The rationale behind the choice for client-server is *separation of concerns* (Hürsch and Lopes, 1995), which works in both directions. On the one hand, the client does not need to offer all services by itself, as this can be delegated to the server. The server, on the other hand, only needs to provide the minimum of information necessary for a client to be able to perform its intended action. On the Web, this translates to Web servers providing resources a browser does not have pre-installed, such as Web pages and images for example. Browsers, on the other hand, are fully responsible for the presentation of the content, so the server does not need to spend time on that (Yeager and McGrath, 1996).

This illustrates how separation of concerns is crucial for *scalability*: the information is distributed, and so is processing power. The presented form of scalability has another dimension, namely time: clients and servers can evolve independently, as long as the contract does not change. For example, while Web servers have been sending HTML for decades, new devices such as mobile phones can still use it—despite a changed presentation concept—because the devices themselves are responsible for rendering the HTML (Berners-Lee *et al.*, 1992).

3.3 Uniform interface constraints

In contrast to the client-server constraints, which are shared among many distributed systems, the uniform interface constraints are defining for the REST architectural style. This uniform interface between components can be seen as the lowest common denominator to access all services provided by servers in the same generic way. This simplifies the architecture and makes client-server messages easy to interpret, since no message is specific to a certain application domain. The REST architectural style defines four constraints that realize the behavior of the uniform interface: *identification of resources*, *resource manipulation through representations*, *self-descriptive messages*, and *hypermedia as the engine of application state*. We will describe each of them in detail, accompanied by an example that does not comply with REST and by an example that does.

3.3.1 Identification of resources

The essential unit of information in REST architectures is a *resource*, a conceptual entity that must be *uniquely identifiable*. On the Web, this means that each resource must have its own URL. As trivial as this might sound, this practice is not applied everywhere. A typical sign of not identifying resources by URLs is if the browser's back button is broken—clicking it does not bring you back to the page you visited previously. Berners-Lee (1998) has stressed the importance of deliberate design to ensure longevity of URLs. More recently, these ideas have been formalized by Sauermann and Cyganiak (2008).

Non-REST compliant A link on a museum Web page brings us to an object of its collection at the URL `http://example.org/collection/showObject.aspx`. If we follow a link to another object, the page's content changes, but the URL remains the same. This means that the information of what object we are viewing is maintained outside of the URL. As a result, we cannot bookmark the URL for later usage, nor can we send it to somebody else.

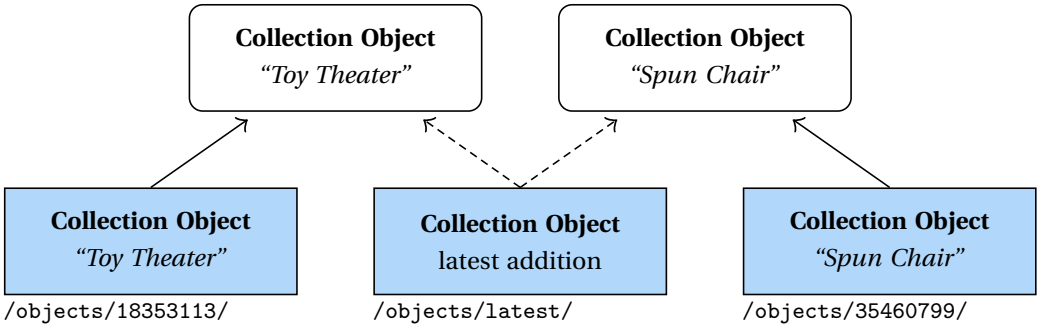

Figure 2: Resources *(colored)* are conceptual mappings, pointing to entities *(white)*. Which entity the resource points to can change and so can the entity itself; the semantics of the mapping, however, cannot.

REST compliant A museum Web page shows a certain object of its collection at the URL `http://example.org/objects/18353113/`. Another object is accessible at `/objects/35460799/`, and similarly, each object has its own URL, which we can share or bookmark for later usage.

The important difference with for instance database systems, is that the URL relation is *conceptual*. This has been illustrated in Figure 2. The entities of our application domain are indicated on top in white; they are objects in a collection and exist independently of the Web application we built on top of it. On the bottom row, resources and their corresponding URL can be seen. For instance, the conceptual mapping *Toy Theater* will always correspond to the object with that name in the collection through the URL `/objects/18353113/`. Interestingly, the conceptual mapping *latest addition* will always point to the most recently acquired object in the collection, but the identity of that concrete object will of course change over time. This indicates the conceptual nature of resources.

3.3.2 Resource manipulation through representations

Now that a shared concept of identification between client and server has been established, the next question is how information can be transfered between them. An essential property of REST architectures is that resources themselves are *not* transfered; instead, client and server exchange a *representation* of a resource. This contrasts with file systems, wherein an identifier *(file name)* always corresponds to a specific physical representation *(the file)*. On the Web, an identifier *(a URL)* corresponds to a conceptual entity *(a resource)*, which can have different representations, depending on the capabilities of the client. For instance, a resource can be represented in HTML for human viewers, and in JSON for consumption by software such as JavaScript applications in the browser.

Non-REST compliant A museum provides access to the HTML version of an object in its collection at `http://example.org/objects/18353113/`. However, the JSON version must be accessed through `http://api.example.org/getObjectJson.php?id=18353113`, and an API key is necessary for all requests.

REST compliant The object is accessible through `http://example.org/objects/18353113/` and, depending on the request, the server replies with HTML or JSON. In the future, RDF might be supported through this same URL.

Note how in the non-compliant example, the identification happens on the technical level instead of the conceptual level. The URL identifies "the HTML representation of object 18353113" instead of "object 18353113". This makes the exchange of URLs between different systems difficult, as the choice for a specific representation is tied to the URL. Furthermore, the addition of new representation formats (such as RDF) would imply that new URLs have to be assigned again, not to mention the complexity of removing support for old representation formats. Without

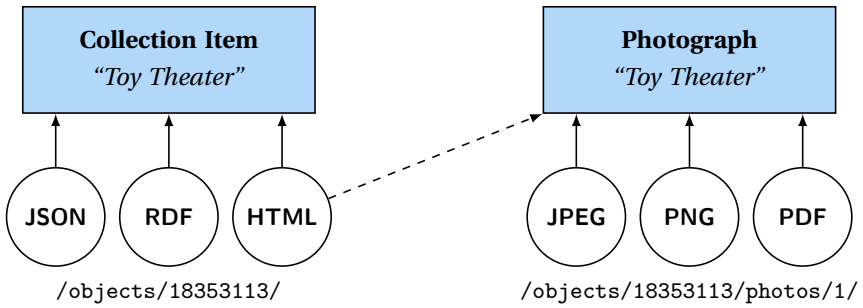

Figure 3: A representation *(circle)* captures the state of a resource *(rectangle)* and consists of data and metadata. Representations can include (links to) other resources, as is the case here with the HTML representation. Note how the target of the link is the *resource* (which is stable) and not any particular *representation* (which can be different).

this constraint, we indeed arrive at a *multi-API*: every type of client needs its own interface to the collection. The purpose of REST is exactly to provide a uniform interface whose contract can be maintained in the longterm.

The concept is illustrated in Figure 3. Every resource can be accessed and manipulated through different representations. The data format of a representation is often referred to as a *media type* (Freed and Borenstein, 1996) and sometimes known as a *hypermedia type* if the document type natively supports hypertext controls such as links (Amundsen, 2011). Because we almost exclusively deal with HTML representations of resources on the Web, many people mistakenly assume a resource can only have a single representation. However, as the figure shows, other representations of the same resource might be served, such as JSON and RDF. Client and server agree on the employed representation by the process of *content negotiation* (Klyne, 1999). In HTTP, this is usually implemented by letting the client specify its media type preferences, which the server then combines with its own preferences to serve the optimal media type for the interaction ("server-driven negotiation", Fielding *et al.*, 1999). An important aspect of representations is that they ease technological transitions, since supporting different types of clients comes down to providing additional representations.

3.3.3 Self-descriptive messages

The uniform interface of REST architectures is considerably simplified by requiring all messages to be self-descriptive. Concretely, this means that every message should contain all information necessary to understand and process it, independent of possible preceding messages (Fielding, 2000).

Non-REST compliant A collection website provides a search function. When searching for "toy", we see the first page of search results `/objects?filter=toy`. To reach the second page, we have to click a button that submits the text `nextpage` to the server.

REST compliant On the first page of search results `/objects?filter=toy`, there is a link to the second page `/objects?filter=toy&page=2`.

In the non-compliant example, we notice the message is not self-descriptive. The text `nextpage` does not fully define the request: the next page of *what* should be shown, and what page is the user currently at? Since this information is not present in the request, it must be maintained somewhere else, but the client does not have any control over that. Hence, he cannot predict what will happen when a `nextpage` request is sent out. In contrast, the compliant example uses messages that fully define the request: the query is `toy` and we require page 2. In order to have self-descriptive messages, interactions must be *stateless*. In general, the client should not assume the server remembered anything about the previous interaction and should therefore resend request details such as resource identifier, authentication details, media type preferences, *etc*.

To contribute to self-descriptiveness, HTTP only defines a limited number of *methods* that may be used in messages, such as `GET` (retrieving a representation of a resource), `POST` (creating or annotating a resource), `PUT` (storing a resource), and `DELETE` (removing a resource). Furthermore, the specification describes whether or not these methods have qualities such as *safeness* and *idempotence* that need to be respected (Davis, 2012). This limited set of methods makes it easy for any party to understand a message by itself, in contrast to programming languages where custom method names can be associated with specific semantics (Van Roy and Haridi, 2004).

3.3.4 Hypermedia as the engine of application state

Closely tied to the concept of statelessness is the question how to transition from one state to another. This REST constraint, known as the *hypermedia constraint* indicates that all state changes should happen through hypermedia. Media types such as HTML natively support controls that allow users to navigate and this constraint mandates that the server inserts controls that lead to next steps a user can take. While this is mostly the case on the human Web, as we are constantly clicking from one place to another, this is far less frequent in representation formats targeted at machines.

Non-REST compliant The JSON representation of a collection object at `http://example.org/objects/35460799/` is:

```
{
  "title": "Spun Chair",
  "producer": {
    "name": "Herman Miller Furniture Company",
    "id": 18049013
  }
}
```

There is no URL in this representation. As such, a client using the JSON representation cannot retrieve more details about the producer through hypermedia, i.e., by following links. Instead, the Web API documentation has to be consulted by a developer to understand how a producer's details can be retrieved through the ID `18049013`.

REST compliant The JSON representation of an object contains the URL to the producer's metadata:

```
{
  "title": "Spun Chair",
  "producer": {
    "name": "Herman Miller Furniture Company",
    "url": "/people/18049013/"
  }
}
```

This allows a client application to access the producer's details without inspecting the documentation, similar to how people navigate HTML documents through links without prior instruction.

In the non-compliant example, we notice the interaction using JSON is *not* driven by hypermedia, but by *out-of-band information* that has to be interpreted separately, even though the interaction using HTML happens fully through hypermedia. Thereby, the Web application gives a different *affordance* (Norman, 1988) to people and software clients: the HTML representation affords navigating to the producer, whereas the JSON representation does not. This increases the production cost of clients, because the documentation is to be consulted at each step, and also makes the contract between a client and a server more fragile—if something changes, the implementation of

the client has to change. In contrast, if the client is following links, then these links can simply be updated without requiring a change in the client code.

In those cases where the interaction is *not* driven by hypermedia, the client often needs to resort to assumptions about the server's URL structure. For instance, it might assume that the producer with internal ID `18049013` can be retrieved by concatenating this ID to the string `/people/`, i.e., `http://example.org/people/18049013`. However, such an assumption endangers the loose coupling between clients and servers, since a server's used URL mechanism is out-of-band information. Such practices implicitly augment the contract between a client and a server with exact knowledge of the latter's URL structure, which is undesired. Berners-Lee captured this in his "Opacity of URIs" Axiom: "*The only thing you can use an identifier for is to refer to an object. When you are not dereferencing, you should not look at the contents of the URI string to gain other information*" (Berners-Lee, 1996). So while the server can put a certain structure in the URLs it offers—as encouraged by Berners-Lee (1998)—the client *cannot* use any information except the URL as a whole. In the past, the Axiom has been misinterpreted as *"URIs must not contain any elements that can be connected to the resource in a meaningful way"* (Panzer, 2008), but this would clearly be in contradiction with other work (Berners-Lee, 1998; Sauermann and Cyganiak, 2008) and current practice on the (Semantic) Web. Even though the majority of URIs *do* contain certain semantically relevant elements, clients should never rely on their presence.

When designing representations, developers often wrongly assume that software clients do not need these affordances. However, the hypermedia principle that any piece of information is linked to other pieces has been crucial to the success of the human Web; we should therefore not underestimate its impact on the Web for software clients. This importance is captured by the following definition: "[hypertext is] the simultaneous presentation of information and controls such that *the information becomes the affordance* through which the user (or automaton) obtains choices and selects actions." (Fielding, 2008, emphasis added). The revolution of the Web is indeed that information has become *actionable*; it is no longer a static piece of text but an interface that affords people and software clients to obtain more information of their choice. In order to enable this powerful mechanism, the fourth and last constraint of the uniform interface thus demands that *any* representation—be it HTML, JSON, or RDF—contains the links that lead to possible next steps. Note that this is also the driver behind Linked Data: any piece of information should link to others that augment its context.

4 A sustainable REST architecture for HTTP servers

4.1 A generic architecture to support change

As REST is an architectural style rather than an actual architecture, several implementations are possible. Our goal is to maximize the longevity of our server implementation and to minimize the amount of effort needed to react to change. Hence, we will discuss a generic architecture that implements the REST architectural style and that is easily extensible for various use cases in which data and metadata have to be exposed in a sustainable way, which is compatible with the current data infrastructure in the cultural heritage domain. The UML diagram in Figure 4 shows the different components and their relation. We will follow the route of a message exchange through this architecture.

Step 1: Sending the request The client creates the request, which consists of a URL that identifies a resource and an HTTP method specifying the action. To the request message, it adds metadata such as its preferences regarding the content type. This request is sent to the HTTP Server through the HTTP protocol.

Step 2: Receiving and parsing the request Inside the server, the *RequestHandler* is listening for incoming requests and receives the *Client*'s HTTP request, which it parses into components: method, URL, and metadata.

Step 3: Identifying the resource The *RequestHandler* asks the *Resolver* to map the URL it extracted from the message to an internal identifier of the resource. This fully decouples the external URL from the internal naming

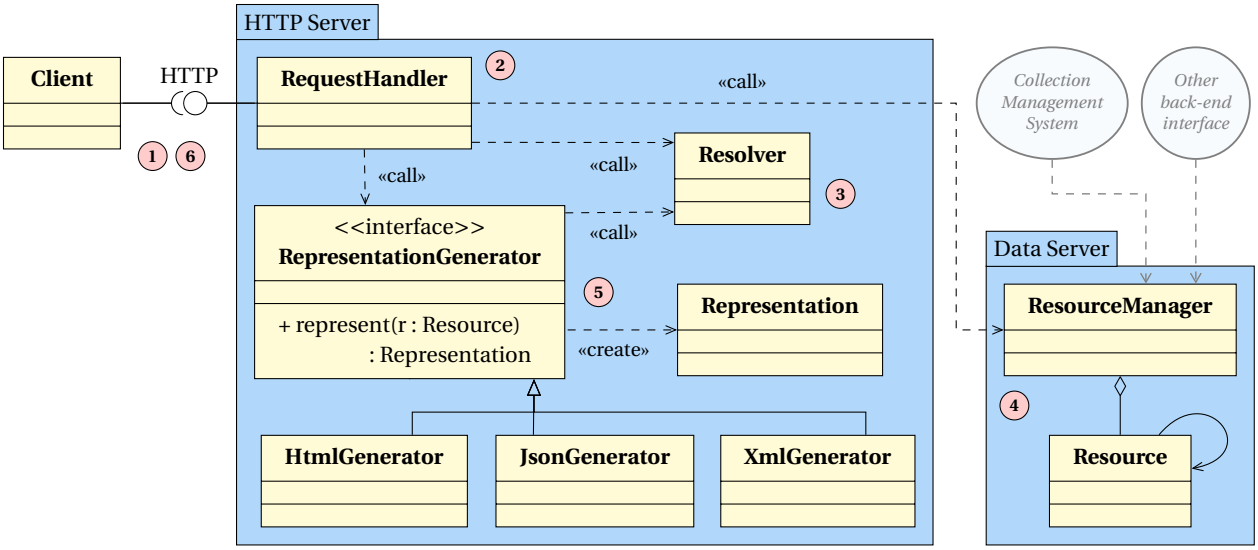

Figure 4: The *Client* exchanges HTTP messages with the *RequestHandler*, the entry point to the *HTTP Server* and interface to the *Data Server*. The multiple implementations of *RepresentationGenerator* provide support for different representations. Internal applications, such as a Collection Management System, continue to access the *Data Server* directly.

scheme, ensuring that both can evolve independently.

Step 4: Retrieving and/or modifying the resource The *RequestHandler* uses the internal resource identifier to retrieve the correct resource from the *ResourceManager*. If the issued HTTP method demands a modification, and the user is permitted to perform it, then the underlying database is updated first.

Step 5: Generating the representation Based on the *Client*'s content type preferences indicated in the HTTP message, the *RequestHandler* selects a *RepresentationGenerator* out of the available generator list (which can be extended). A specific generator, for instance the *HtmlGenerator*, takes the *Resource* and serializes it as a *Representation* in a specific format. If the *Resource* contains references to other *Resource*s, their internal identifiers are converted to URLs by the *Resolver* and then embedded in the representation.

Step 6: Sending the representation The *RepresentationGenerator* sends the representation back to the *Client*. The *Client* can now follow links in this representation, in response to which the cycle starts again at Step 1.

4.2 The impact of change

The essential aspect in the described architecture is the *decoupling* of components to avoid the propagation of change. We will now investigate different change scenarios and see how they impact the system.

First, note that current applications on the back-end, such as a Collection Management System, are unaffected: they continue to interface with the *Data Server*. The *HTTP Server* acts as an intermediary between the *client* and the *Data Server*, shielding off the implementation details of the latter. If a new *Data Server* is chosen to replace the current system, the *Client* is still able to issue HTTP requests as before. In other words, URLs point to the same resources, regardless of how they are stored at the server. The *RequestHandler* will have to be adapted to the new *Data Server*, and possibly the *Resolver* will convert identifiers in a different way. No other components are affected: this major change can be dealt with by relatively minor adaptations in existing components. Most importantly, no *Client* implementations (website scripts, applications…) have to change, so independent evolution of client and server is possible.

Supporting new types of clients can happen by developing a new *RepresentationGenerator*. Figure 4 already shows generators for HTML, JSON, and XML. Suppose we want to offer support for Linked Data by offering RDF, then only an *RdfGenerator* has to be created and plugged into the system. No other changes are required. Compare this to the multi-API paradigm, in which it would be necessary to build an entirely new API in order to support RDF. The development costs would be much higher, plus documentation and support would need to be provided. Instead, we reuse the existing API and architecture. Importantly, URLs remain the same as well, again avoiding changes on the client. Clients continue to indicate their preferences, and if the RDF format is a match, then the *RdfGenerator* is automatically selected by the *RequestHandler*.

These two scenarios indicate that the architecture is resilient to change. Because there is no direct coupling between URLs, representations, and the internal data store, the system can gradually grow. This is a requirement for a system in an environment that is steadily evolving under constant technological change.

5 Use case: implementing REST at the Cooper-Hewitt Museum

5.1 The Cooper-Hewitt collection and its collection management database

The Cooper-Hewitt National Design Museum was founded at the end of the 19th century and harbors a collection of historical and contemporary design. In 2011, the museum closed for a 3-year long renovation, rethinking the collection in terms of different narratives for different visitor groups. A major part of this transformation is to remain true to its mandate to be there for every one: both the expert and the general public. This is translated into a technical challenge: the collection and its metadata must be offered in flexible ways inside and outside of the museum. Not only is the agility to rapidly fulfill emerging user demands a requirement, it is also the ambition of the museum to promote the discovery of new unforeseen requirements.

Over the years, Cooper-Hewitt has used TMS as collection management software in combination with eMuseum to drive the public front-end of the collection. This software environment does not offer any of the amount of flexibility necessary for the information flow and agility described above. The museum needs to move faster than allowed by the internal architecture of TMS and the dictates of eMuseum's pricing model. As a generic software solution marketed towards a broad range of cultural heritage institutions, TMS is designed to be very flexible towards individual exceptions in the data model, at the cost of a rather complex relational database design. The problem is that this makes the data non-transparent and thus difficult to approach from outside TMS.

Despite all of its shortcomings, a pragmatic choice was made to keep TMS for the curatorial tasks it has been performing over the years and by doing so the database continues to serve as the system responsible for metadata creation and management. However, in order to address the agility concerns and to have maximum flexibility on the front-end website, eMuseum is abandoned. Data is exported from TMS into a limber environment which can be in an alpha release perpetually. The first goal therefore was to generate an automatically generated CSV (Comma-Separated Value) export from TMS. The export is then ingested in a PHP web-application framework which gives the freedom to rapidly build interfaces and indexes, custom-tailored to needs which can evolve rapidly.

This agile approach allowed the museum to bypass the complexities of data modeling. One conceptual requirement which became paramount however is the issuing of persistent identifiers and making almost everything a first-class resource on the Web. In other words, as many facets as possible of the collection should be expressed as URLs. For example `http://collection.cooperhewitt.org/countries/france` and `http://collection.cooperhewitt.org/objects/colors/ff1493/` respectively give access to all objects manufactured in France or which have deep pink (represented by the color code `ff1493`) as their dominant color palette.

Persistent identifiers under the form of URLs also are fundamental if Cooper Hewitt wants to realize its vision of holding hand with as many partners as possible on the Web. The museum has been actively reconciling its

metadata with authorities such as VIAF (Virtual Authority Files) but also with knowledge bases like Wikipedia and Freebase. For example, the museum does not have a biography of the American designer Ray Eames, but it provides through the reconciliation process a link to his wikipedia page. Other cultural heritage institutions, such as the MoMA (Museum of Modern Art) are also actively reconciling their metadata with the same knowledge bases and authorities. This approach makes it then feasible at a second stage to make it explicit what artists, places, etc. are shared across institutions.

The following sections will demonstrate how REST principles were used to implement this infrastructure and to guarantee a stable environment for issuing persistent URLs.

5.2 Use of REST principles to publish the collection

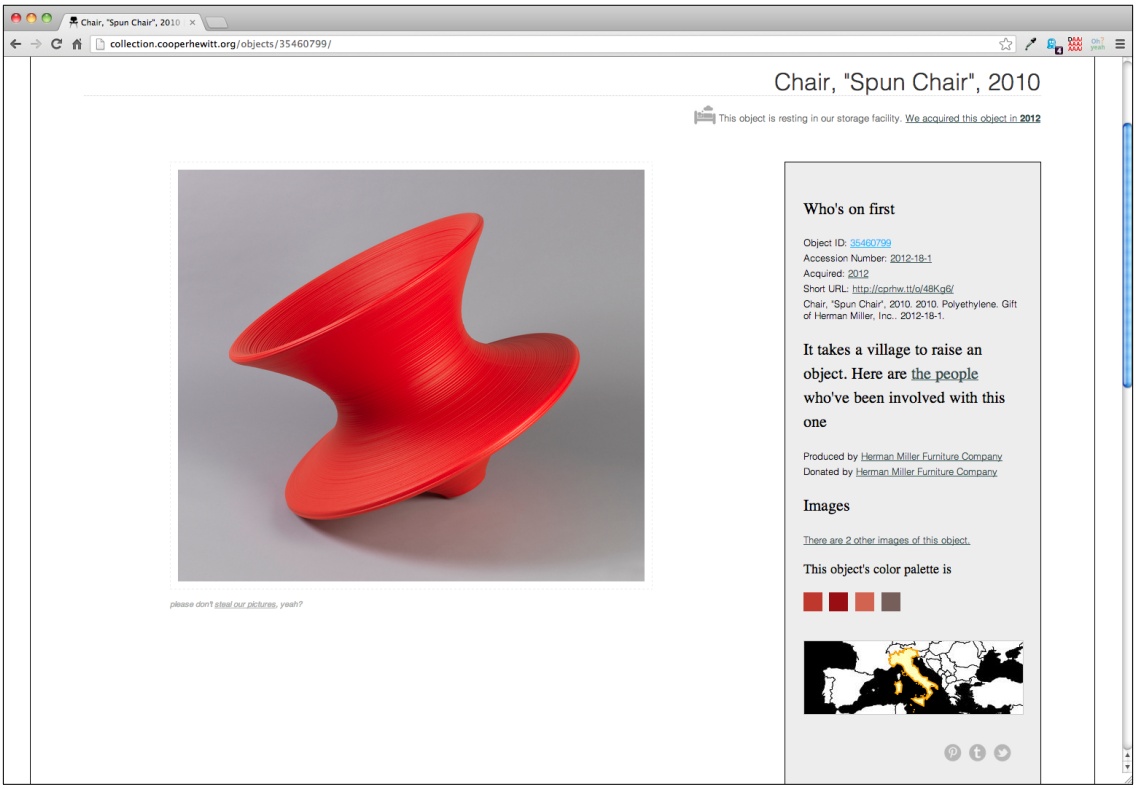

Figure 5: An HTML representation of a collection item on the Cooper-Hewitt website. Following the REST principles, the JSON representation is available at the same URL through content negotiation.

Properly designing a URI space has an profound impact on the persistence of identifiers (Berners-Lee, 1998). Careful design ensures that the server can maintain a URI in the long-term. The URI space for the Cooper-Hewitt collection has been chosen as follows. The base URL for is `http://collection.cooperhewitt.org/`, which is on a different subdomain then the main website `http://www.cooperhewitt.org/`. This easily separates the content management system of the main website, which focuses on a different type of content, from the collection. Furthermore, this limits the number of URIs that should be accounted for, as all non-collection-related information (such as museum news and events) are allocated in a different URI space. Within the collection, `/objects/35460799/` is an example of an object URL. People and organizations involved with objects in various roles are accessible through URLs such as `/people/18049013/`. In addition to these two main types of resources, *collection* resources also occur: lists of objects. For instance, `/people/18049013/objects/manufacturer/` lists all objects where the organization with ID 18049013 is the manufacturer. Conversely, `/objects/35460799/people/`

lists all people and organizations involved in the object with ID 35460799. This makes it straightforward for humans to understand the URL structure, but also for a *Resolver* to extract the necessary bits of information. For instance, the URI pattern `/people/{personid}/objects/{relationname}` can capture the structure of the former and pass the `personid` and `relationname` values to a *ResourceManager*. As indicated in Section 3.3.4, knowledge about this structure *only* resides on the server side; all URLs are opaque to the client (Berners-Lee, 1996). Instead, the client should use hypermedia controls supplied by the server to navigate from one resource to another.

For the public collection website, the HTML representation is of central importance. Based on a PHP templating engine, the properties of the retrieved resource are embedded in a generic structure that is consistent with the rest of the website. The HTML representation of a collection links to related resources such as people and organizations, and embeds an image of the item if available.

As the website also needs to provide an interface to Web applications, machine-interpretable representations are also needed. Furthermore, the Cooper-Hewitt wants to use their own API inside the museum, to avoid the duplicate effort of making another interface to TMS. This refers to both back-end operations of registrars and curators, as well as public-facing interactive media and experiences inside the galleries and future exhibitions. Having an evolvable, machine-accessible interface on the public website was therefore a logical decision. The JSON format has been chosen because of its simple integration with Web applications, the native language of which is JavaScript. A `JsonGenerator` fulfills this role. In accordance with the REST principles, the JSON representations are accessible through the same resource URI. In order to test the implementation of the REST principles, the command line tool cURL can be used. By executing the following instructions in command line:
`curl http://collection.cooperhewitt.org/objects/35460799/ -H "Accept: text/html"`,
we receive an HTML representation. If we ask for JSON:
`curl http://collection.cooperhewitt.org/objects/35460799/ -H "Accept: application/json"`,
then we will receive a machine-readable representation of the same resource. This means that the same URL can be exchanged between different systems, regardless of whether the consumer of this URL is a human or a machine. Consequently, any operation performed on or with this resource (such as adding annotations) will have the same effect for all involved parties.

Should the Cooper-Hewitt decide to support Linked Data in the future as well, then it will merely be a matter of creating a new generator, for instance for RDF/Turtle. The same URLs will continue to work; and machine clients that can make use of the extra semantics of RDF will be able to retrieve the RDF representation and perhaps act in more complex ways with it than with the JSON representation. For instance, since RDF uses URI identifiers internally, it might prove interesting to follow those URIs and try to find more information about the referenced concepts.

6 Conclusions

This article brought attention to the relevance of REST principles for the LIS discipline. More in particular, we focused on how the REST architectural style can contribute to the longevity of URLs, which are playing a central role within the application of Linked Data principles and the automated usage of documents and their metadata.

We started out by precisely defining a Web API and indicated that today's strategy of building new APIs to support each individual technological change is not a desirable longterm strategy. The conceptual introduction to the client-server and uniform interface constraints underlined the importance of providing people and software clients with equal affordance on information in a transparent way, allowing information to be exchanged between different parties regardless of a concrete format. Based on this, a general architecture was presented with the help of a UML diagram. This schematic outline demonstrated how the HTTP Server functions as a buffer between the Client and the Data Server. The decoupling allows both parties to evolve independently and to avoid the propagation of change.

In order to illustrate the practical benefits of this information architecture, a use case was presented based on the collection database of the Cooper-Hewitt National Design Museum. In the context of the current renovation of the museum, a fundamental rethinking of how the museum is providing access to its objects and metadata took

place. For strategic reasons, it was decided not to invest in a migration or a remodeling of the current collection registration database. TMS will be used over the years to come for metadata creation and management, but an automated export is pulled from TMS into a PHP web-application framework. This approach gives the freedom to rapidly build custom interfaces for needs which might evolve rapidly. However, in this agile context there is a need to serve and maintain persistent identifiers for objects and different facets of the objects. The URLs play a central role in creating connections with other institutions and providing the user community the opportunity to build services on top of these identifiers. The conceptual idea of serving representations (and not resources) is demonstrated with an example from the use case. By doing so, the case study illustrated how an institution can make use of REST to facilitate both human and machine consumption, without investing in multiple APIs.

Confronted with the current economic downturn, many libraries, archives and museums simply not have the luxury to spend years on the development of the perfect software, meeting all needs for the documentation of their resources. The pragmatic approach of the Cooper-Hewitt museum to feed an automated export from a less than perfect collection management database into a PHP web-application framework is a valuable example for both practitioners and researchers. Through the use of REST principles, the museum manages to decouple the access provided to its objects and metadata from the specific technology used. HTTP is the interface to the Web, and when the REST principles are applied, it is the only interface any application needs. The conceptual introduction and the practical use case hopefully can serve as an example for other cultural heritage organizations and stimulate further research in the use of the REST architectural style by other LIS researchers.

References

Amundsen, M. (2011), "Hypermedia types", in Wilde, E. and Pautasso, C. (Eds.), *REST: From Research to Practice*, Springer, New York, NY, pp. 93–116.

Berners-Lee, T. (1991), "The original HTTP as defined in 1991", available at http://www.w3.org/Protocols/HTTP/AsImplemented.html (accessed 23 July 2013).

Berners-Lee, T. (1996), "Universal Resource Identifiers – Axioms of Web Architecture", available at http://www.w3.org/DesignIssues/Axioms.html (accessed 23 July 2013).

Berners-Lee, T. (1998), "Cool URIs don't change", available at http://www.w3.org/Provider/Style/URI.html (accessed 23 July 2013).

Berners-Lee, T., Cailliau, R., Groff, J. F. and Pollermann, B. (1992), "World-Wide Web: The information universe", *Electronic Networking: Research, Applications and Policy*, Vol. 1, Meckler, Westport, CT, pp. 52–58.

Bizer, C., Heath, T. and Berners-Lee, T. (2009), "Linked Data – the story so far", *International Journal On Semantic Web and Information Systems*, Vol. 5, pp. 1–22.

Boydens, I. and van Hooland, S. (2011), "Hermeneutics applied to the quality of empirical databases", *Journal of Documentation*, Vol. 67, pp. 279–289.

Davis, C. (2012), What if the web were not RESTful?, in *Proceedings of the Third International Workshop on RESTful Design*, ACM, New York, NY, pp. 3–10.

Fielding, R. T. (2000), Architectural Styles and the Design of Network-based Software Architectures, PhD thesis, University of California, Irvine, California.

Fielding, R. T. (2008), "REST APIs must be hypertext-driven", available at http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (accessed 23 July 2013).

Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999), "Hypertext Transfer Protocol – HTTP/1.1", available at http://www.ietf.org/rfc/rfc2616.txt (accessed 23 July 2013).

Flyvbjerg, B. (2006), "Five misunderstandings about case-study research", *Qualitative Inquiry*, Vol. 12, pp. 219–245.

Freed, N. and Borenstein, N. (1996), "Multipurpose Internet Mail Extensions (MIME) part one: Format of Internet messsage bodies", available at http://tools.ietf.org/html/rfc2045 (accessed 23 July 2013).

Gomes, D. and Silva, M. J. (2006), Modelling information persistence on the web, in *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, ACM, New York, NY, USA, pp. 193–200.
URL: *http://doi.acm.org/10.1145/1145581.1145623*

Guy, M. (2009), JISC Good APIs Management Report, Technical report, JISC.

Heidegger, M. (1954), "Die Frage nach der Technik", *Vorträge und Aufsätze*, Vol. 8, Verlag Günther Neske, Pfullingen, pp. 13–55.

Hürsch, W. and Lopes, C. (1995), Separation of concerns, Technical report, College of Computer Science, Northeastern University, Boston, MA.

Khare, R. and Taylor, R. (2004), Extending the Representational State Transfer (REST) architectural style for decentralized systems, in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pp. 428–437.

Klyne, G. (1999), "Protocol-independent content negotiation framework", available at http://www.ietf.org/rfc/rfc2703.txt (accessed 23 July 2013).

Kroski, E. (2008), "On the move with the mobile Web: Libraries and mobile technologies", *Library Technology Reports*, Vol. 44, American Library Association, Chicago, IL.

McCown, F., Chan, S., Nelson, M. L. and Bollen, J. (2005), "The availability and persistence of web references in D-Lib magazine", *CoRR*, Vol. abs/cs/0511077.

Norman, D. A. (1988), *The Design of Everyday Things*, Doubleday, New York.

Panzer, M. (2008), Cool URIs for the DDC: towards web-scale accessibility of a large classification system, in *Proceedings of the International Conference on Dublin Core and Metadata Applications, Berlin*, pp. 183–190.

Pautasso, C., Zimmermann, O. and Leymann, F. (2008), RESTful Web services vs. "Big" Web services: making the right architectural decision, in *Proceedings of the 17th international conference on World Wide Web*, WWW '08, ACM, New York, NY, pp. 805–814.

Powell, A. (2005), A service oriented view of the JISC Information Environment, Technical report, UKOLN.

Ramsay, S. and Rockwell, G. (2012), *Debates in the Digital Humanities*, Minesota Press, chapter Developing things: notes towards an epistemology of building in the digital humanities, pp. 75–84.

Sauermann, L. and Cyganiak, R. (2008), "Cool URIs for the Semantic Web", available at http://www.w3.org/TR/cooluris/ (accessed 23 July 2013).

Severance, C. (2012), "Discovering JavaScript Object Notation", *Computer*, Vol. 45, pp. 6–8.

Sinha, A. (1992), "Client-server computing", *Communications of the ACM*, Vol. 35, ACM, New York, NY, pp. 77–98.

Summers, E., Isaac, A., Redding, C. and Krech, D. (2008), LCSH, SKOS and Linked Data, in *Proceedings of the International Conference on Dublin Core and Metadata Applications, Berlin*, pp. 25–33.

Summers, E. and Salo, D. (2013), Linking things on the web: A pragmatic examination of linked data for libraries, museums and archives, Technical report, Unpublished working paper, available on http://arxiv.org/abs/1302.4591.

van Hooland, S., Verborgh, R. and de Walle, R. V. (2012), "Joining the linked data cloud in a cost-effective manner", *Information Standards Quarterly*, Vol. 24, pp. 24–29.

Van Roy, P. and Haridi, S. (2004), *Concepts, Techniques, and Models of Computer Programming*, MIT Press, Cambridge, USA.

Verborgh, R., Mannens, E. and Van de Walle, R. (2013), The rise of the Web for Agents, in *Proceedings of the First International Conference on Building and Exploring Web Based Environments*, pp. 69–74.

Vinoski, S. (2007), "REST eye for the SOA Guy", *Internet Computing, IEEE*, Vol. 11, pp. 82–84.

Wilde, E. (1999), *Wilde's WWW: Technical Foundations of the World Wide Web*, Springer, New York, NY.

Yeager, N. J. and McGrath, R. E. (1996), *Web server technology: the advanced guide for World Wide Web information providers*, Morgan Kaufmann, San Fransisco, CA.

Zuzak, I. and Schreier, S. (2012), "Arrested development: Guidelines for designing rest frameworks", *IEEE Internet Computing*, Vol. 16, IEEE Computer Society, Los Alamitos, CA, USA, pp. 26–35.