Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 15, 2016

A. Keranen
Ericsson
M. Kovatsch
ETH Zurich
K. Hartke
Universitaet Bremen TZI

March 14, 2016

RESTful Design for Internet of Things Systems
draft-keranen-t2trg-rest-iot-01

Abstract

This document gives guidance for designing Internet of Things (IoT) systems that follow the principles of the Representational State Transfer (REST) architectural style.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 15, 2016.

Copyright Notice

Table of Contents

1. Introduction

The Representational State Transfer (REST) architectural style [REST] is a set of guidelines and best practices for building distributed hypermedia systems.

When REST principles are applied to the design of a system, the result is often called RESTful and in particular an API following these principles is called a RESTful API.

Different protocols can be used with RESTful systems, but at the time of writing the most common protocols are HTTP [RFC7230] and CoAP [RFC7252].

RESTful design facilitates many desirable features for a system, such as good scaling properties. RESTful APIs are also often simple and lightweight and hence easy to use also with various IoT applications. The goal of this document is to give basic guidance for designing RESTful systems and APIs for IoT applications and give pointers for more information.

Design of a good RESTful IoT system has naturally many commonalities with other Web systems. Compared to other systems, the key characteristics of many IoT systems include:

- data formats, interaction patterns, and other mechanisms that minimize, or preferably avoid, the need for human interaction
- preference for compact and simple data formats to facilitate efficient transfer over (often) constrained networks and lightweight processing in constrained nodes

2. Terminology

This section explains some of the common terminology that is used in the context of RESTful design for IoT systems. For terminology of constrained nodes and networks, see [RFC7228].

Application State: The state kept by a client between requests. This typically includes the "current" resource, the set of active requests, the history of requests, bookmarks (URIs stored for later retrieval) and application-specific state.

Cache: A local store of response messages and the subsystem that controls storage, retrieval, and deletion of messages in it.

Client: A node that sends requests to servers and receives responses.

Content Negotiation: The practice of determining the "best" representation for a client when examining the current state of a resource. The most common forms of content negotiation are Proactive Content Negotiation and Reactive Content Negotiation.

Form: A hypermedia control that enables a client to change the state of a resource.

Forward Proxy: An intermediary that is selected by a client, usually via local configuration rules, and that can be tasked to make requests on behalf of the client. This may be useful, for example, when the client lacks the capability to make the request itself or to service the response from a cache in order to reduce response time, network bandwidth and energy consumption.

Gateway: See "Reverse Proxy".

Hypermedia Control: A component embedded in a representation that describes a future request, such as a link or a form. By

performing the request, the client can change resource state and/ or application state.

Idempotent Method: A method where multiple identical requests with that method lead to the same visible resource state as a single such request. For example, the PUT method replaces the state of a resource with a new state; replacing the state multiple times with the same new state still results in the same state for the resource. However, the response from the server can be different when the same idenpotent method is used multiple times. For example when DELETE is used twice on an existing resource, the first request would remove the association and return success acknowledgement whereas the second request would likely result in error response due to non-existing resource.

Link: A hypermedia control that enables a client to navigate between resources and thereby change the application state.

Media Type: A string such as "text/html" or "application/json" that is used to label representations so that it is known how the representation should be interpreted and how it is encoded.

Method: An operation associated with a resource. Common methods include GET, PUT, POST, and DELETE (see Section 3.5 for details).

Origin Server: A server that is the definitive source for representations of its resources and the ultimate recipient of any request that intends to modify its resources. In contrast, intermediaries (such as proxies caching a representation) can assume the role of a server, but are not the source for representations as these are acquired from the origin server.

Proactive Content Negotiation: A content negotiation mechanism where the server selects a representation based on the expressed preference of the client. For example, in an IoT application, a client could send a request with preferred media type "application/senml+json".

Reactive Content Negotiation: A content negotiation mechanism where the client selects a representation from a list of available representations. The list may, for example, be included by a server in an initial response. If the user agent is not satisfied by the initial response representation, it can request one or more of the alternative representations, selected based on metadata (e.g., available media types) included in the response.

Representation Format: A set of rules for encoding information in a sequence of bytes. In the Web, the most prevalent representation

format is HTML. Other common formats include plain text (in UTF-8 or another encoding) and formats based on JSON [RFC7159] or XML. With IoT systems, often compact formats based on JSON, CBOR [RFC7049], and EXI [W3C.REC-exi-20110310] are used.

Representation: A sequence of bytes, plus representation metadata, that captures the current or intended state of a resource and that can be transferred between clients and servers (possibly via one or more intermediaries).

Representational State Transfer (REST): An architectural style for Internet-scale distributed hypermedia systems.

Resource: An item of interest identified by a URI. Anything that can be named can be a resource. A resource often encapsulates a piece of state in a system. Typical resources in an IoT system can be, e.g., a sensor, the current value of a sensor, the location of a device, or the current state of an actuator.

Resource State: A mapping of a resource to a set of values that may change over time.

Reverse Proxy: An intermediary that appears as a server towards the client but satisfies the requests by forwarding them to the actual server (possibly via one or more other intermediaries). A reverse proxy is often used to encapsulate legacy services, to improve server performance through caching, and to enable load balancing across multiple machines.

Safe Method: A method that does not result in any state change on the origin server when applied to a resource. For example, the GET method only returns a representation of the resource state but does not change the resource. Thus, it is always safe for a client to retrieve a representation without affecting server-side state.

Server: A node that listens for requests, performs the requested operation and sends responses back to the clients.

Uniform Resource Identifier (URI): A global identifier for resources. See Section 3.4 for more details.

3. Basics

3.1. Architecture

The components of a RESTful system are assigned one of two roles: client or server. User agents are always in the client role and have the initiative to issue requests. Intermediaries (such as forward proxies and reverse proxies) implement both roles, but only forward requests to other intermediaries or origin servers. They can also translate requests to different protocols, for instance, as CoAP-HTTP cross-proxies.

Note that the terms "client" and "server" refer only to the roles that the nodes assume for a particular message exchange. The same node might act as a client in some communications and a server in others.

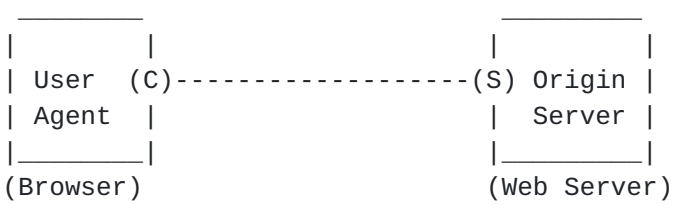


Figure 1: Client-Server Communication

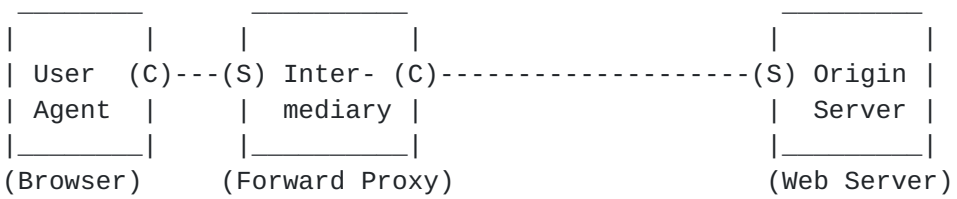


Figure 2: Communication with Forward Proxy

Reverse proxies are usually imposed by the origin server. In addition to the features of a forward proxy, they can also provide an interface for non-RESTful services such as legacy systems or alternative technologies such as Bluetooth ATT/GATT. This property is enforced by the layered system constraint of REST, which says that a client cannot see beyond the server it is connected to.

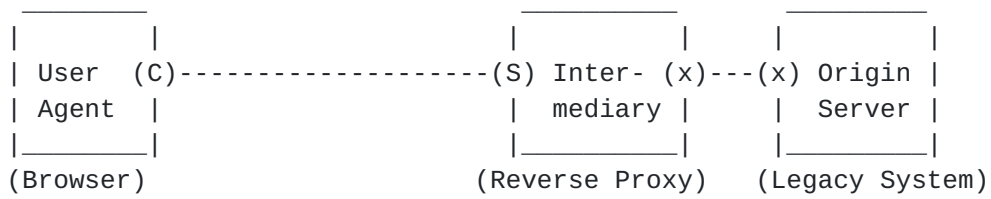


Figure 3: Communication with Reverse Proxy

Nodes in IoT systems often implement both roles. Unlike intermediaries, however, they can take the initiative as a client (e.g., to register with a directory, such as CoRE Resource Directory [I-D.ietf-core-resource-directory], or to interact with another thing) and act as origin server at the same time (e.g., to serve sensor values or provide an actuator interface).

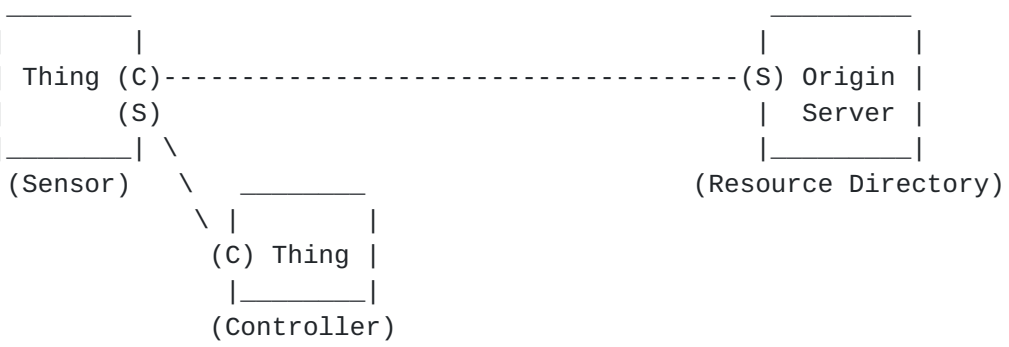


Figure 4: Constrained RESTful environments

3.2. System design

When designing a RESTful system, the state of the distributed application must be assigned to the different components. Here, it is important to distinguish between "session state" and "resource state".

Session state encompasses the control flow and the interactions between the components (see Section 2). Following the statelessness constraint, the session state must be kept only on clients. On the one hand, this makes requests a bit more verbose since every request must contain all the information necessary to process it. On the other hand, this makes servers efficient, since they do not have to keep any state about their clients. Requests can easily be distributed over multiple worker threads or server instances. For the IoT systems, it lowers the memory requirements for server implementations, which is particularly important for constrained servers and servers serving large amount of clients.

Resource state includes the more persistent data of an application (i.e., independent of the application control flow). This can be static data such as device descriptions, persistent data such as system configuration, but also dynamic data such as the current value of a sensor on a thing.

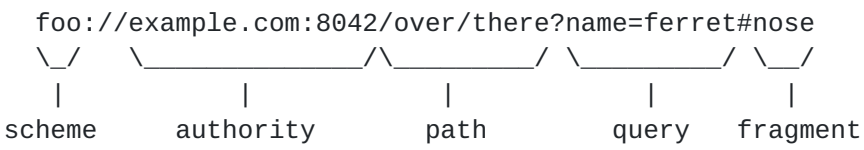
3.3. Resource modeling

Important part of RESTful API design is to model the system as a set of resources whose state can be retrieved and/or modified and where resources can be potentially also created and/or deleted.

Resource representations have a media type that tells how the representation should be interpreted. Typical media types for IoT systems include "text/plain" for simple UTF-8 text, "application/octet-stream" for arbitrary binary data, "application/json" for JSON [RFC7159], "application/senml+json" [I-D.jennings-core-senml] for Sensor Markup Language (SenML) formatted data, "application/cbor" for CBOR [RFC7049], "application/exi" for EXI [W3C.REC-exi-20110310]. Full list of registered internet media types is available at the IANA registry [IANA-media-types] and media types registered for use with CoAP are listed at CoAP Content-Formats IANA registry [IANA-CoAP-media].

3.4. Uniform Resource Identifiers (URIs)

Uniform Resource Identifiers (URIs) are used to indicate a resource for interaction, to reference a resource from another resource, to advertise or bookmark a resource, or to index a resource by search engines.



A URI is a sequence of characters that matches the syntax defined in [RFC3986]. It consists of a hierarchical sequence of five components: scheme, authority, path, query, and fragment (from most significant to least significant). A scheme creates a namespace for resources and defines how the following components identify a resource within that namespace. The authority identifies an entity that governs part of the namespace, such as the server "www.example.org" in the "http" scheme. A host name (e.g., a fully qualified domain name) or an IP address, potentially followed by a transport layer port number, are usually used in the authority component for the "http" and "coap" schemes. The path and query contain data to identify a resource within the scope of the URI's scheme and naming authority. The path is hierarchical; the query is non-hierarchical. The fragment allows to refer to some portion of the resource, such as a section in an HTML document.

For RESTful IoT applications, typical schemes include "https", "coaps", "http", and "coap". These refer to HTTP and CoAP, with and

without Transport Layer Security (TLS) [RFC5246]. (CoAP uses Datagram TLS (DTLS) [RFC6347], the variant of TLS for UDP.) These four schemes also provide means for locating the resource; using the HTTP protocol for "http" and "https", and with the CoAP protocol for "coap" and "coaps". If the scheme is different for two URIs (e.g., "coap" vs. "coaps"), it is important to note that even if the rest of the URI is identical, these are two different resources, in two distinct namespaces.

The query parameters can be used to parametrize the resource. For example, a GET request may use query parameters to request the server to send only certain kind data of the resource (i.e., filtering the response). Query parameters in PUT and POST requests do not have such established semantics and are not commonly used.

3.5. HTTP/CoAP Methods

Section 4.3 of [RFC7231] defines the set of methods in HTTP; Section 5.8 of [RFC7252] defines the set of methods in CoAP. The following lists the most relevant methods and gives a short explanation of their semantics.

3.5.1. GET

The GET method requests a current representation for the target resource. Only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a response to GET.

A payload within a GET request message has no defined semantics.

The GET method is safe and idempotent.

3.5.2. POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics.

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server sends a 201 (Created) response containing a Location header field that provides an identifier for the resource created and a representation that describes the status of the request while referring to the new resource(s).

The POST method is not safe nor idempotent.

3.5.3. PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent.

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

The PUT method is not safe, but is idempotent.

3.5.4. DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server.

The DELETE method is not safe, but is idempotent.

3.6. HTTP/CoAP Status/Response Codes

Section 6 of [RFC7231] defines a set of Status Codes in HTTP that are used by application to indicate whether a request was understood and satisfied, and how to interpret the answer. Similarly, Section 5.9 of [RFC7252] defines the set of Response Codes in CoAP.

The status codes consist of three digits (e.g., "404" with HTTP or "4.04" with CoAP) where the first digit expresses the class of the code. Implementations do not need to understand all status codes, but the class of the code must be understood. Codes starting with 1 are informational; the request was received and being processed. Codes starting with 2 indicate successful request. Codes starting with 3 indicate redirection; further action is needed to complete the

request. Codes stating with 4 and 5 indicate errors. The codes starting with 4 mean client error (e.g., bad syntax in request) whereas codes starting with 5 mean server error; there was no apparent problem with the request but server was not able to fulfill the request.

Responses may be stored in a cache to satisfy future, equivalent requests. HTTP and CoAP use two different patterns to decide what responses are cacheable. In HTTP, the cacheability of a response depends on the request method (e.g., responses returned in reply to a GET request are cacheable). In CoAP, the cacheability of a response depends on the response code (e.g., responses with code 2.04 are cacheable). This difference also leads to slightly different semantics for the codes starting with 2; for example, CoAP does not have a 2.00 response code whereas 200 ("OK") is commonly used with HTTP.

4. Security Considerations

This document does not define new functionality and therefore does not introduce new security concerns. However, security consideration from related specifications apply to RESTful IoT design. These include:

- HTTP security: Section 9 of [RFC7230], Section 9 of [RFC7231], etc.
- CoAP security: Section 11 of [RFC7252]
- URI security: Section 7 of [RFC3986]

5. Acknowledgement

The authors would like to thank Mert Ocak, Heidi-Maria Back, Tero Kauppinen, and Michael Koster for the reviews and feedback.

6. References

6.1. Normative References

[I-D.ietf-core-resource-directory]
Shelby, Z., Koster, M., Bormann, C., and P. Stok, "CoRE Resource Directory", draft-ietf-core-resource-directory-05 (work in progress), October 2015.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. Dissertation, University of California, Irvine , 2000.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[W3C.REC-exi-20110310] Schneider, J. and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0", World Wide Web Consortium Recommendation REC-exi-20110310, March 2011, <http://www.w3.org/TR/2011/REC-exi-20110310>.

6.2. Informative References

[I-D.jennings-core-senml] Jennings, C., Shelby, Z., Arkko, J., and A. Keranen, "Media Types for Sensor Markup Language (SENML)", draft-jennings-core-senml-04 (work in progress), January 2016.

[IANA-CoAP-media] "CoAP Content-Formats", n.d., <http://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats>.

[IANA-media-types]
"Media Types", n.d., <http://www.iana.org/assignments/media-types/media-types.xhtml>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <http://www.rfc-editor.org/info/rfc7228>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

Appendix A. Future Work

- More details on the definition of application state. Is server involved and to what extent.
- Discuss design patterns, such as "Observing state (asynchronous updates) of a resource", "Executing a Function", "Events as State", "Conversion", "Collections", "robust communication in network with high packet loss", "unreliable (best effort) communication", "3-way commit", etc.
- Discuss directories, such as CoAP Resource Directory
- More information on how to design resources; choosing what is modeled as a resource, etc.

Authors' Addresses

Ari Keranen
Ericsson
Jorvas 02420
Finland

Email: ari.keranen@ericsson.com

Matthias Kovatsch
ETH Zurich
Universitaetstrasse 6
Zurich CH-8092
Switzerland

Email: kovatsch@inf.ethz.ch

Klaus Hartke
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

Email: hartke@tzi.org