Take control of your APIs

Two layers. Unlimited possibilities.

3scale's unique separation of the cloud management layer and traffic control mechanism delivers unmatched flexibility and performance. Deploy on premise or in the cloud, and manage access control policy, documentation, rate limits, and more from a unified cloud dashboard.

Create your free account ›

3scale

Learn more at 3scale.net

REST: Foundations of RESTful Architecture

By Brian Sletten

INTRODUCTION

The Representational State Transfer (REST) architectural style is not a technology you can purchase or a library you can add to your software development project. It is first and foremost a worldview that elevates information into a first class element of the architectures we build.

The ideas and terms we use to describe "RESTful" systems were introduced and collated in Dr. Roy Fielding's thesis, "Architectural Styles and the Design of Network-based Software Architectures". This document is academic and uses formal language, but remains accessible and provides the basis for the practice.

The summary of the approach is that by making specific architectural choices, we can elicit desirable properties from the systems we deploy. The constraints detailed in this architectural style are not intended to be used everywhere, but they are widely applicable.

The concepts are well demonstrated in a reference implementation we call the Web. Advocates of the REST style are basically encouraging organizations to apply the same principles within their boundaries as they do to external facing customers with web pages.

THE BASICS

A RESTful service is exposed through a Uniform Resource Locator (URL). This is a logical name that separates the identity of the resource from what is accepted or returned. The URL scheme is defined in RFC 1738.

A sample RESTful URL might be something like the following fake API for a library:

```
http://fakelibrary.org/library
```

What is actually exposed is not necessarily an arbitrary service, however, but an information resource representing something of value to a consumer. The URL functions as a handle for the resource, something that can be requested, updated, or deleted.

This starting point would be published somewhere as the way to begin interacting with the library's REST services. What is returned could be XML, JSON or—more appropriately—a hypermedia format such as Atom or a custom MIME type. The general guidance is to reuse existing formats where possible, but there is a growing tolerance for properly designed media types.

To request the resource, a client would issue a Hypertext Transfer Protocol (HTTP) GET request to retrieve it. This is what happens when you type a URL into a browser and hit return, select a bookmark, or click through an anchor reference link.

For programmatic interaction with a RESTful API, any of a dozen or more client side APIs or tools could be used. To use the curl command line tool, you could type something like:

```
$ curl http://fakelibrary.org/library
```

This will return the default representation on the command line. You may not want the information in this form, however. Fortunately, HTTP has a mechanism by which you can ask for information in a different form. By specifying an "Accept" header in the request, if the server supports that representation, it will return it. This is known as content negotiation and is one of the more underused aspects of HTTP. Again, using curl, this could be done with:

```
$ curl -H "Accept:application/json"
http://fakelibrary.org/library
```

This ability to ask for information in different forms is possible because of the separation of the name of the resource from its form. The 'R' in REST is 'representation', not 'resource'. Keep this in mind and build systems that allow clients to ask for information in the forms they want. We will revisit this topic later.

Possible URLs for our fake library might include:

- `http://fakelibrary.org/library` - general information about the library and the basis for discovering links to search for specific books, DVDs, etc.
- `http://fakelibrary.org/book` - an "information space" for books. Conceptually, it is a placeholder for all possible books. Clearly, if it were resolved, we would not want to return all possible books, but it might perhaps return a way to discover books through categories, keyword search, etc.
- `http://fakelibrary.org/book/category/1234` - within the information space for books, we might imagine browsing them based on particular categories (e.g. adult fiction, children's books, gardening, etc.) It might make sense to use the Dewey Decimal system for this, but we can also imagine custom groupings as well. The point is that this "information space" is potentially infinite and driven by what kind of information people will actually care about.
- `http://fakelibrary.org/book/isbn/978-0596801687` - a reference to a particular book. Resolving it should include information about the title, author, publisher, number of copies in the system, number of copies available, etc.

These URLs mentioned above will probably be read-only as far as the library patrons are concerned, but applications used by librarians might actually manipulate these resources.

For instance, to add a new book, we might imagine POSTing an XML representation to the main /book information space. In curl, this might look like:

```
$ curl -u username:password -d @book.xml -H "Content-type:
text/xml" http://fakelibrary.org/book
```

At this point, the resource on the server might validate the results, create the data records associated with the book and return a 201 response code indicating that a new resource has been created. The URL for the new resource can be discovered in the Location header of the response.

An important aspect of a RESTful request is that each request contains enough state to answer the request. This allows for the conditions of visibility and statelessness on the server, desirable properties for scaling systems up and identifying what requests are being made. This helps to enable the caching of specific results. The combination of a server's address and the state of the request combine to form a computational hash key into a result set:

```
http://fakelibrary.org + /book/isbn/978-0596801687
```

Because of the nature of the GET request (discussed later), this allows a client to make very specific requests, but only if necessary. The client can cache a result locally, the server can cache it remotely or some intermediate architectural element can cache it in the middle. This is an application-independent property that can be designed into our systems.

Just because it is possible to manipulate a resource does not mean everyone will be able to do so. We can absolutely put a protection model in place that requires users to authenticate and prove that they are allowed to do something before we allow them to. We will have some pointers on ways of securing RESTful services at the end of this card.

WHAT ABOUT SOAP?

What about it? There is a false equivalence asserted about REST and SOAP that yields more heat than light when they are compared. They are not the same thing. They are not intended to do the same thing even though you can solve many architectural problems with either approach.

The confusion largely stems from the confused idea that REST "is about invoking Web Services through URLs." That has about as much truth to it as the idea that "agile methodologies are about avoiding documentation." Without a deeper understanding of the larger goals of an approach, it is easy to lose the intent of the practices.

REST is best used to manage systems by decoupling the information that is produced and consumed from the technologies that produce and consume it. We can achieve the architectural properties of:

- Performance
- Scalability
- Generality
- Simplicity
- Modifiability
- Extensibility

This is not to say SOAP-based systems cannot be built demonstrating some of these properties. But SOAP is best leveraged when the lifecycle of a request cannot be maintained in the scope of a single transaction because of technological, organizational, or procedural complications.

RICHARDSON MATURITY MODEL

In part to help elucidate the differences between SOAP and REST, and to provide a framework for classifying the different kinds of systems many people were inappropriately calling "REST," Leonard Richardson introduced a Maturity Model. You can think of the classifications as a measure of how closely a system embraces the different pieces of Web Technology: Information resources, HTTP as an application protocol, and hypermedia as the medium of control.

LEVEL	ADOPTION
0	This is basically where SOAP is. There are no information resources, HTTP is treated like a transport protocol, and there is no concept of hypermedia. Conclusion: REST and SOAP are different approaches.

LEVEL	ADOPTION
1	**URLs are used**, but not always as appropriate information resources, and everything is usually a GET request (including requests that update server state). Most people new to REST first build systems that look like this.
2	**URLs are used** to represent information resources. **HTTP is respected** as an application protocol, sometimes including content negotiation. Most Internet-facing "REST" web services are really only at this level because they only support non-hypermedia formats.
3	**URLs are used** to represent information resources. **HTTP is respected** as an application protocol including content negotiation. **Hypermedia** drives the interactions for clients.

Calling it a "maturity model" might seem to suggest that you should only build systems at the most "mature" level. That should not be the take-home message. There is value at being at Level 2, and the shift to Level 3 is often simply the adoption of a new MIME type. The shift from Level 0 to Level 3 is much harder, so even incremental adoption adds value.

Start by identifying the information resources you would like to expose. Adopt HTTP as an application protocol for manipulating these information resources—including support for content negotiation. Then, when you are ready, adopt hypermedia-based MIME types and you should get the full benefits of REST.

VERBS

The limited number of verbs in RESTful systems confuses and frustrates people new to the approach. What seem like arbitrary and unnecessary constraints are actually intended to encourage predictable behavior in non-application-specific ways. By explicitly and clearly defining the behavior of these verbs, clients can be self-empowered to make decisions in the face of network interruptions and failure.

There are four main HTTP verbs (sometimes called methods) used by well-designed RESTful systems.

GET

The most common verb on the Web, a GET request transfers representations of named resources from a server to a client. The client does not necessarily know anything about the resource it is requesting. What it gets back is a bytestream tagged with metadata that indicates how the client should interpret it. On the Web, this is typically "text/html" or "application/xhtml+xml". As we indicated above, using content negotiation, the client can be proactive about what is requested as long as the server supports it.

One of the key points about the GET request is that it should not modify anything on the server side. It is fundamentally a safe request. This is one of the biggest mistakes made by people new to REST. With RMM Level 1 systems, you often see URLs such as:

```
http://example.com/res/action=update?data=1234
```

Do not do this! Not only will RESTafarians mock you, but you will not build RESTful ecosystems that yield the desired properties. The safety of a GET request allows it to be cached.

GET requests are also intended to be **idempotent**. This means that issuing a request more than once will have no consequences. This is an important property in a distributed, network-based infrastructure. If a client is interrupted while it is making a GET request, it should be empowered to issue it again because of the idempotency of the verb. This is an enormously important point. In a well-designed infrastructure, it does not matter what the client is requesting from which application. There will always be application-specific behavior, but the more we can push into non-application-specific behavior, the more resilient and easier to maintain our systems will be.

POST

The situation gets a little less clear when we consider the intent of the POST and PUT verbs. Based on their definitions, both seem to be used to create or update a resource from the client to the server. They have distinct purposes, however.

POST is used when the client cannot predict the identity of the resource it is requesting to be created. When we hire people, place orders, submit forms, etc., we cannot predict how the server will name these resources we are creating. This is why we POST a representation of the resource to a handler (e.g. servlet). The server will accept the input, validate it, verify the user's credentials, etc. Upon successful processing, the server will return a 201 HTTP response code with a "Location" header indicating the location of the newly created resource.

Note: Some people treat POST like a conversational GET on creation requests. Instead of returning a 201, they return a 200 with the body of the resource created. This seems like a shortcut to avoid a second request, but it also conflates POST and GET and complicates the potential for caching the resource. Try to avoid the urge to take shortcuts at the expense of the larger picture. It seems worth it in the short-term, but over time, these shortcuts will add up and likely work against you.

Another major use of the POST verb is to "append" a resource. This is an incremental edit or a partial update, not a full resource submission. For that, use the PUT operation. A POST update to a known resource would be used for something like adding a new shipping address to an order or updating the quantity of an item in a cart.

Because of this partial update potential, POST is **neither** safe nor idempotent.

A final common use of POST is to submit queries. Either a representation of a query or URL-encoded form values are submitted to a service to interpret the query. It is usually fair to return results directly from this kind of a POST since there is no identity associated with the query.

Note: Consider turning a query like this into an information resource itself. If you POST the definition into a query information space, you can then issue GET requests to it, which can be cached. You can also share this link with others.

PUT

Many developers largely ignore the PUT verb because HTML forms do not currently support it. It serves an important purpose, however, and is part of the full vision for RESTful systems.

A client can issue a PUT request to a known URL as a means of passing the representation back to the server in order to do an overwrite action. This distinction allows a PUT request to be **idempotent** in a way that POST updates are not.

If a client is in the process of issuing a PUT overwrite and it is interrupted, it can feel empowered to issue it again because an overwrite action can be reissued with no consequences; the client is attempting to control the state, so it can simply reissue the command.

Note: This protocol-level handling does not necessarily preclude the need for higher (application-level) transactional handling, but again, it is an architecturally desirable property to bake in below the application level.

PUT can also be used to create a resource if the client is able to predict the resource's identity. This is usually not the case, as we discussed under the POST section, but if the client is in control of the server-side information spaces, it is a reasonable thing to allow.

DELETE

The DELETE verb does not find wide use on the public Web (thankfully!), but for information spaces you control, it is a useful part of a resource's lifecycle.

DELETE requests are intended to be **idempotent**, so you should generally build resources that respond to DELETE requests by failing silently and returning a 204 (No Content) even if the resource has already been deleted. Some security policies may require you to return a 404 for non-existent or deleted resources so DELETE requests do not leak information about the presence of resources.

There are three other verbs that are not as widely used but provide value.

HEAD

The HEAD verb is used to issue a request for a resource without actually retrieving it. It is a way for a client to check for the existence of a resource and possibly discover metadata about it.

OPTIONS

The OPTIONS verb is also used to interrogate a server about a resource by asking what other verbs are applicable to the resource.

PATCH

The newest of the verbs, PATCH was only officially adopted as part of HTTP in early 2010. The goal is to provide a standardized way to express partial updates. Because POST can be used for anything, it is unclear when it is being used for partial updates.

A PATCH request in a standard format could allow an interaction to be more explicit about the intent. There are RFCs from the IETF for patching XML and JSON.

If the client issues a PATCH request with an If-Match header, it is possible for this partial update to become **idempotent**. An interrupted request can be retried because, if it succeeded the first time, the If-Match header will differ from the new state. If they are the same, the original request was not handled and the PATCH can be applied.

RESPONSE CODES

HTTP response codes give us a rich dialogue between clients and servers about the status of a request. Most people are only familiar with 200, 403, 404 and maybe 500 in a general sense, but there are many more useful codes to use. The tables presented here are not comprehensive, but cover many of the most important codes you should consider using in a RESTful environment.

The first collection of response codes indicates that the client request was well formed and processed. The specific action taken is indicated by one of the following:

CODE	DESCRIPTION
200	**OK**. The request has successfully executed. Response depends upon the verb invoked.
201	**Created**. The request has successfully executed and a new resource has been created in the process. The response body is either empty or contains a representation containing URIs for the resource created. The Location header in the response should point to the URI as well.

CODE	DESCRIPTION
202	**Accepted**. The request was valid and has been accepted but has not yet been processed. The response should include a URI to poll for status updates on the request. This allows asynchronous REST requests
204	**No Content**. The request was successfully processed but the server did not have any response. The client should not update its display.

Table 1 - *Successful Client Requests*

CODE	DESCRIPTION
301	**Moved Permanently**. The requested resource is no longer located at the specified URL. The new Location should be returned in the response header. Only GET or HEAD requests should redirect to the new location. The client should update its bookmark if possible.
302	**Found**. The requested resource has temporarily been found somewhere else. The temporary Location should be returned in the response header. Only GET or HEAD requests should redirect to the new location. The client need not update its bookmark as the resource may return to this URL.
303	**See Other**. This response code has been reinterpreted by the W3C Technical Architecture Group (TAG) as a way of responding to a valid request for a non-network addressable resource. This is an important concept in the Semantic Web when we give URIs to people, concepts, organizations, etc. There is a distinction between resources that can be found on the Web and those that cannot. Clients can tell this difference if they get a 303 instead of 200. The redirected location will be reflected in the Location header of the response. This header will contain a reference to a document about the resource or perhaps some metadata about it.

Table 2 - *Redirected Client Requests*

The third collection of response codes indicates that the client request was somehow invalid and will not be handled successfully if reissued in the same condition. These failures include potentially improperly-formatted requests, unauthorized requests, requests for resources that do not exist, etc.

CODE	DESCRIPTION
400	**Bad Request.**
401	**Unauthorized.**
403	**Forbidden.**
404	**Not Found.**
405	**Method Not Allowed.**
406	**Not Acceptable.**
410	**Gone**
411	**Length Required.**
412	**Precondition Failed.**
413	**Entity Too Large.**
414	**URI Too Long.**
415	**Unsupported Media Type.**
417	**Expectation Failed.**

Table 3 - *Invalid Client Requests*

The final collection of response codes indicates that the server was temporarily unable to handle the client request (which may still be invalid) and that it should reissue the command at some point in the future.

CODE	DESCRIPTION
500	**Internal Service Error.**
501	**Not Implemented.**
503	**Service Unavailable.**

Table 4 - *Server Failed to Handle the Request*

The service zones have different scalability requirements according to their function.

REST RESOURCES

THESIS

Dr. Fielding's thesis, "Architectural Styles and the Design of Network-based Software Architectures" is the main introduction to the ideas discussed here: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

RFCS

The specifications for the technologies that define the most common uses of REST are driven by the Internet Engineering Task Force (IETF) Request for Comments (RFC) process. Specifications are given numbers and updated occasionally over time with new versions that obsolete existing ones. At the moment, here are the latest relevant RFCs.

URI:
The generic syntax of URIs as a naming scheme are covered in RFC 3986. A URI is a naming scheme that can include encoding other naming schemes such as website addresses, namespace-aware sub-schemes, etc.

Site: http://www.ietf.org/rfc/rfc3986.txt

URL:
A Uniform Resource Locator (URL) is a form of URI that has sufficient information embedded within it (access scheme and address usually) to resolve and locate the resource.

Site: http://www.ietf.org/rfc/rfc1738.txt

IRI:
An Internationalized Resource Identifier (IRI) is conceptually a URI encoded in Unicode to support characters from the languages of the world in the identifiers they use on the Web. The IETF chose to create a new standard rather than change the URI scheme itself to avoid breaking existing systems and to draw explicit distinctions between the two approaches. Those who support IRIs do so deliberately. There are mapping schemes defined for converting between IRIs and URIs as well.

Site: http://www.ietf.org/rfc/rfc3987.txt

HTTP:
The Hypertext Transfer Protocol (HTTP) version 1.1 defines an application protocol for manipulating information resources generally represented in hypermedia formats. While it is an application-level protocol, it is generally not application specific, and important architectural benefits emerge as a result. Most people think of HTTP and the Hypertext Markup Language (HTML) as "The Web", but HTTP is useful in the development of non-document-oriented systems as well.

Site: http://www.ietf.org/rfc/rfc2616.txt

PATCH Formats:
JavaScript Object Notation (JSON) Patch

Site: https://www.ietf.org/rfc/rfc6902.txt

XML Patch

Site: https://www.ietf.org/rfc/rfc7351.txt

DESCRIPTION LANGUAGES

There is strong interest in having languages to describe APIs to make it easier to document or possibly even generate skeletons for clients and servers. Some of the more popular or interesting languages are described below:

RAML:
A YAML/JSON language for describing Level 2-oriented APIs. It includes support for reusable patterns and traits that can help standardize features across API design.

Site: http://raml.org

Swagger:
Another YAML/JSON language for describing Level 2-oriented APIs. It includes code generators, an editor, visualization of API documentation, and the ability to integrate with other services.

Site: http://swagger.io

Apiary.io:
A collaborative, hosted site with support for Markdown-based documentation of APIs, social interactions around the design process, and support for mock hosted implementations to make it easy to test APIs before they are implemented.

Site: http://apiary.io

Hydra-Cg:
A Hypermedia description language expressed via standards such as JSON-LD to make it easy to support Linked Data and interaction with other data sources.

Site: http://www.hydra-cg.com

IMPLEMENTATIONS

There are several libraries and frameworks available for building systems that produce and consume RESTful systems. While any web server can be configured to supply a REST API, these frameworks, libraries, and environments make it easier to do so.

Here is an overview of some of the main environments:

JAX-RS:
This specification adds support for REST to JEE environments.

Site: https://jax-rs-spec.java.net

Restlet:
The Restlet API was one of the first attempts at creating a Java API for producing and consuming RESTful systems. The attention paid to both the client and server sides of the equation yields some very clean and powerful APIs.

The Restlet Studio is a free tool that allows conversion between RAML and Swagger-based API descriptions, as well as skeleton and stub support for Restlet, Node, and JAX-RS servers and clients.

Site: http://restlet.org

NetKernel:
One of the more interesting RESTful systems, NetKernel represents a microkernel-based environment supporting a wide variety of architectural styles. It benefits from the adoption of the economic properties of the Web in a software architecture. You can think of it as "bringing REST inside." Whereas any REST-based system kind of looks the same externally, NetKernel continues to look like that within its execution environment as well.

Site: http://netkernel.org

Play:
One of the two main Scala REST frameworks.

Site: https://www.playframework.com

Spray:
One of the two main Scala REST frameworks. This is designed

to work with the Akka actor model.

Site: http://spray.io

Express:
One of the two main Node.js REST frameworks.

Site: http://expressjs.com

hapi:
One of the two main Node.js REST frameworks.

Site: http://hapijs.com

Sinatra:
Sinatra is a domain specific language (DSL) for creating RESTful applications in Ruby.

Site: http://www.sinatrarb.com

OpenRasta:
OpenRasta brings the concept of REST to the .NET platform in ways that allow it to be deployed alongside ASP.NET and WCF components.

Site: http://openrasta.org

There are many other implementations to investigate. For more information, please consult this list of known implementations:

http://code.google.com/p/implementing-rest/wiki/RESTFrameworks

BOOKS

"**RESTful Web APIs**" by *Leonard Richardson*, *Mike Amundsen* and *Sam Ruby*, 2013. O'Reilly Media.

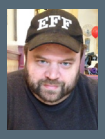
"**RESTful Web Services Cookbook**" by *Subbu Allamaraju*, 2010. O'Reilly Media.

"**REST in Practice**" by *Jim Webber*, *Savas Parastatidis* and *Ian Robinson*, 2010. O'Reilly Media.

"**Restlet in Action**" by *Jerome Louvel* and *Thierry Boileau*, 2011. Manning Publications.

"**Resource-Oriented Architecture Patterns for Webs of Data (Synthesis Lectures on the Semantic Web: Theory and Technology)**" by *Brian Sletten*, 2013. Morgan & Claypool.

ABOUT THE AUTHOR



Brian Sletten is a liberal arts-educated software engineer with a focus on forward-leaning technologies. His experience has spanned many industries including retail, banking, online games, defense, finance, hospitality, publications and health care. He has a B.S. in Computer Science from the College of William and Mary and lives in Auburn, CA. He focuses on web architecture, resource-oriented computing, social networking, the Semantic Web, data science, 3D graphics, visualization, scalable systems, security consulting and other technologies of the late 20th and early 21st Centuries. He is also a rabid reader, devoted foodie and has excellent taste in music. If pressed, he might tell you about his International Pop Recording career.

RECOMMENDED BOOK

This cookbook includes more than 100 recipes to help you take advantage of REST, HTTP, and the infrastructure of the Web. You'll learn ways to design RESTful web services for client and server applications that meet performance, scalability, reliability, and security goals, no matter what programming language and development framework you use.

BUY NOW

CREDITS:
Editor: **G. Ryan Spain** | Designer: **Farhin Dorothi** | Production: **Chris Smith** | Sponsor Relations: **Brandon Rosser**

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0 **$7.95**