FOUNDATIONS OF

RESTful Architecture

WRITTEN BY BRIAN SLETTEN, PRESIDENT, BOSATSU CONSULTING
UPDATED BY CHASE DOELLING DIRECTOR OF ALLIANCES MARKETING AT CLOUD ELEMENTS

CONTENTS

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. REST is a worldview that elevates information into a first-class element of the architectures we build.

The ideas and terms used 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 is an academic document but is a convenient and comprehensible way to explain the basis of RESTful architecture.

The summary of the approach is that by making specific architectural choices, we can obtain desirable properties from the systems we create. The constraints detailed in this architectural style are widely applicable; however, they are not intended to be used universally.

Due to the Web's prolific impact on consumer preferences, advocates of the REST style are encouraging organizations to apply the same principles within their boundaries as they do to external-facing customers with web pages. This Refcard will cover the basic constraints and properties found in modern REST web implementations.

THE BASICS

What does Representational State Transfer mean? **Transferring**, accessing, and manipulating textual data **representations** in a **stateless** manner. When deployed correctly, it provides a uniform interoperability, between different applications on the internet. The term stateless is crucial piece to this as it allows applications to communicate agnostically. A RESTful API service is exposed through a Uniform Resource Locator (URL). This logical name separates the identity of the resource from what is accepted or returned. The URL scheme is defined in RFC 1738, which can be found here: ietf.org/rfc/rfc1738.txt

An example of a RESTful URL is shown in this mock API library:

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

What is actually exposed is not necessarily an arbitrary service, but an information resource representing something of value to a consumer. The URL must have the capability of being created, requested, updated, or deleted. This sequence of actions is commonly referred to as CRUD.

This starting point would be published somewhere as a way to begin interacting with the library's REST services. The request, when returned, could be XML, JSON, or a hypermedia format. It's recommended to reuse existing formats when possible, but there is a growing tolerance for properly designed media types to be used.

To request and retrieve the resource, a client would issue a Hypertext Transfer Protocol (HTTP) GET request. This is the most common request and is executed every time you type a URL into a browser and hit return, select a bookmark, or click through an anchor reference link.

REST Architecture

An Integration Cheat Sheet

Brought to you by:

cloud elements

GENERAL

- RESTful service is exposed through a Uniform Resource Locator (URL)
- The URL functions as a handle for the resource, which can be requested, updated, or deleted
- REST returns a XML, JSON or a hypermedia format such as Atom or a custom MIME type
- Issue a Hypertext Transfer Protocol (HTTP) GET request to retrieve the data, which is done when you return a URL in the browser
- Programmatic interaction with a RESTful API, use a client side API/tool
  Example (curl): **$ curl http://fakelibrary.org/library**
- Content negotiation, specify an "Accept" header for the data format
  Example: **$ curl –H "Accept:application/json" http://fakelibrary.org/library**

PROPERTIES

- Performance
- Simplicity
- Visibility
- Reliability
- Scalability
- Modifiability
- Portability

CONSTRAINTS

- Uniform interface
- Code on demand
- Layered system
- Client-server
- Stateless
- Cacheable

REST vs. SOAP

- REST accesses data
- SOAP performs operations

CODE ERROR & DESCRIPTION

1XX	Informational	200	Executed	302	Temp. Found	401	Unauthorized	410	Gone	415	No Media Type
2XX	Success	201	Created	303	See Other	403	Forbidden	411	Length Req.	417	Failed
3XX	Redirection	202	Accepted	304	Not Modified	404	Not Found	412	Precondition	500	Internal Error
4XX	Client Error	204	No Content	307	Temp. Redirect	405	Not a Method	413	Entity Too Big	501	Not implemented
5XX	Server Error	301	Permanent Move	400	Bad Request	406	Not Accepted	414	URI Too Long	503	Unavailable

VERBS

GET	Returns an entity in response to the requested resource.	POST	Returns an entity describing the outcome of the action.
PUT	Stores a new or updated entity at a URI (Uniform Resource Identifier).	DELETE	Request that a resource is removed now or later.
HEAD	Returns the entity's header field related to the requested resource	OPTIONS	Interrogates a server about a resource by asking what other verbs are applicable.
PATCH	Updates only the specified fields of an entity at a URI.		

Every action in the Cloud Element platform is powered by our 100% RESTful APIs. We expose those APIs enabling you to automate and extend your integrations to be fully customizable.

Turn integration into your biggest **competitive advantage.**

LEARN MORE

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

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

This will return the default representation on the command line, however, you may not want the information in this form. Fortunately, HTTP has a built-in mechanism to filter and return information in a different format. If the server supports an "Accept" representation, you are able to specify this in the header and the information in this format. This is known as content negotiation and is one of the more underused aspects of HTTP. This can be done using a curl command similar to the previous example:

```
$ 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. Although the "R" in REST is "representation," not "resource," this should be kept in mind when building systems that allow clients to ask for information in the forms they want. Possible URLs for our example library we might include are:

- 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 useful information about the title, author, publisher, number of copies in the system, number of copies available, etc.

The URLs mentioned above, will probably be read-only as far as the library patrons are concerned, but applications used by librarians should have the added ability to manipulate these resources.

For instance, to add a new book, we might POST an XML representation to the main /book information space. An example using curl looks 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 validates the results, creates the data records associated with the book, and returns 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 visibility and statelessness on the server, desirable properties for scaling systems up, and identifying what requests are being made. This state also enables 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 like this example:

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

The GET request, which will be discussed later, allows a client to make very specific requests, but only when necessary. The client can cache a result locally, the server can cache a result remotely or some intermediate architectural element can cache a result 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, doesn't mean everyone has the capability to do so. By putting a protection model in place that requires users to authenticate and prove that they are allowed to do something, before we give them permission.

WHAT ABOUT SOAP?

There is a false equivalence asserted about REST and SOAP that yields more difficulties than advantages when they are compared. Plain and simple, they are not the same thing. Even though you can solve many architectural problems with either approach, they are not intended to be used interchangeably.

The confusion largely stems from the misunderstood idea that REST "is about invoking Web Services through URLs." This idea is far from the point of the functionalities of RESTful architecture. Without a deeper understanding of the larger picture that RESTful architecture achieves, 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. 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 explain the differences between SOAP and REST, while providing 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.
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" suggests that you should only build systems at the most "mature" level possible. That is not the case, there is value at being at each level, depending on what you need from your requests. The shift from Level 2 to Level 3 is often simply the adoption of a new MIME type. On the other hand, 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, adopt hypermedia-based MIME types and the resulting architecture will provide you the full benefits of REST.

VERBS

Verbs are the methods or actions that are available to interact with the resources on the server. The limited number of verbs in RESTful systems confuses and frustrates people new to the approach. What seems like arbitrary and unnecessary constraints, are in fact 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 failures.

There are four main HTTP verbs which are used by well-designed RESTful systems.

GET

A GET request is the most common verb on the Web. A GET request transfers representations of named resources from a server to a client. Although, the client does not necessarily know anything about the resource it is requesting, the request returns a byte stream tagged with metadata, indicating how the client should interpret the resource. This is typically represented on the web by "text/html" or "application/xhtml+xml." As we indicated above, the client can use content negotiation to 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. You might encounter URLs such as:

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

Do not do this! This will confuse the RESTful ecosystem that you're building, as 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 a critical 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. In well-designed infrastructures, it doesn't matter what the client is requesting from which application. There will always be application-specific behavior, but the more we can push into nonapplication-specific behavior, the more resilient, accessible, and easier to maintain our systems will be.

POST

The situation becomes less clear when we consider the intent of the POST and PUT verbs. Based on their definitions, both seem to be used to add a resource from the client to the server, however, they have distinct purposes.

POST is used when the client cannot predict the identity of the resource that is requested to be created. When we hire people, place orders, submit forms, etc., we cannot predict how the server will name the resources we create. 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 combines POST and GET functions while complicating the potential for caching the resource. 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 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 similar to 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 values are submitted to a service to interpret the query. It is usually fair to return results directly from this kind of POST call 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 a GET request that can be cached. Additionally, you can 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, the client can issue a PUT 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, it's 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 previously 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 is not widely used 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. A DELETE request may be interrupted by a network failure. Whether the request was successfully handled on the first request or not, the resource should respond with a 204 (No Content) response code. It may take some extra handling to keep track of previously deleted resources and resources that never existed (which should return a 404 response code). Some security policies may require you to return a 404 response code for non-existent and deleted resources to prevent leaking 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 the resource. 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. This allows for developers to better understand how to interact and develop against resources.

PATCH

The newest of the verbs, PATCH was only officially adopted as part of HTTP in 2010. The goal is to provide a standardized way to express partial updates. A PATCH request in a standard format allows an interaction to be more explicit about the intent. This is the reason one would use PATCH over POST, even though POST can be used for anything. 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 they cover many of the most important codes you should consider using in a RESTful environment. Each set of numbers can be categorized as the following:

1XX: Informational

2XX: Success

3XX: Redirection

4XX: Client Error

5XX: Server Error

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.
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.
302	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.
304	Not Modified.
307	Temporary Redirect.
308	Permanent Redirect.

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. Client error or perceived as a client error that needs corrected.
401	Unauthorized. Lacking the necessary authorization permissions.

403	Forbidden. The server understands but forbids sending a response.
404	The most famous error code, Not Found. Nothing to see here, try something else.
405	Method Not Allowed. The method used is not supported by the origin.
406	Not Acceptable.
410	Gone. Used instead of a 404 to state that a resource is intentionally missing.
411	Length Required.
412	Precondition Failed.
413	Entity Too Large. The payload is beyond what the server will send
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 Server Error.
501	Not Implemented.
502	Bad Gateway. The server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.
503	Service Unavailable.
504	Gateway Timeout.
505	HTTP Version Not Supported. Likely not secure enough.
508	Loop detected. Server quit the request to prevent an infinite loop.

***Table 4** - Server Failed to Handle the Request*

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

Note: Try the response code 418 and it will return a short but stout response, "I'm a teapot."

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: 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 most 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: 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: 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: ietf.org/rfc/rfc3987.txt

HTTP

The Hypertext Transfer Protocol (HTTP) 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. Version 1.1 Site: ietf.org/rfc/rfc2616.txt
Version 2 Site: httpwg.org/specs/rfc7540.html

PATCH FORMATS:

JavaScript Object Notation (JSON) Patch Site: ietf.org/rfc/rfc6902.txt
XML Patch Site: 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:

OPENAPI SPECIFICATION

The OpenAPI Specification, originally known as the Swagger Specification, is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. SITE: openapis.org

OPEN DATA PROTOCOL

Released by Microsoft is built for the consumption and quiring of RESTful APIs. Sharing some similarities with JDBC and ODBC in that its not limited to relational databases. SITE: odata.org

GOOGLE CLOUD ENDPOINTS

Using the same underlying technologies used across Google for consumer products. Focused on the management of APIs like authorization, deployments and integration. SITE: cloud.google.com/endpoints

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: 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: swagger.io

API BLUEPRINT

Focusing on C++ through Node.js and C# implementations, API Blueprint uses Markdown as its format. Making it easy to adopt and share through GitHub. However, it lacks some higher level tooling.

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: 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: 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:

JAVA BASED

JAX-RS:

This specification adds support for REST to JEE environments. Site: jax-rs-spec.java.net

TALEND - RESTLET:

The Restlet API was one of the first attempts at creating a Java API for producing and consuming RESTful systems. Site: restlet.org

JAVASCRIPT & NODE.JS BASED

EXPRESS:

The most popular Node.js REST frameworks. Site: expressjs.com

HAPI:

One of the two main Node.js REST frameworks. Site: hapijs.com

SCALA BASED

PLAY

Popular for Java and Scala REST frameworks. Site: playframework.com

SPRAY:

Designed to work with the Akka actor model. Site: spray.io

RUBY BASED

SINATRA:

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

PHP BASED

SLIM

A micro PHP framework supporting HTTP routing. Site: slimframework.com

PYTHON BASED

FLASK

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. Site: github.com/pallets/flask

CLIENTS

It is possible to use a browser to invoke a REST API, but there are other types of clients that are useful for testing and building resource-oriented systems.

CURL

One of the more popular libraries and command line tools, curl allows you to invoke various protocols on various resources. Site: curl.haxx.se

HTTPIE

httpie is a very flexible and easy to use client for interacting with resources via HTTP. Site: httpie.org

POSTMAN

Full-blown API testing requires the ability to capture and replay requests, use various authentication and authorization schemes and more. The previous command line tools allow for much of this but Postman is a newer desktop application that makes these activities easier for teams of developers. Site: getpostman.com

PAW

Built for Mac, Paw is a full-featured HTTP client that lets you test and describe the APIs you build or consume. Site: paw.cloud

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 API Design Rulebook: Designing Consistent RESTful Web Service Interfaces" by Mark Masse, 2011. O'Reilly Media.
- "RESTful API Design" by Matthias Biehl, 2016. CreateSpace Publishing.
- "REST in Practice" by Jim Webber, Savas Parastatidis and Ian Robinson, 2010. O'Reilly Media.
- "Restlet in Action" by Jerome Louvel, Thierry Templier, and Thierry Boileau, 2012. Manning Publications.
- "Developing RESTful Services with JAX-RS 2.0, Websockets, and JSON" by Masoud Kalali, 2013. Packt Publishing.

Written by **Chase Doelling**, Director of Alliances Marketing, Cloud Elements

He is a Colorado native, certified beer judge, and purveyor of doughnuts. Chase is a frequent speaker on API and integration topics that focus on making the best use, of best practices, in a very real world. More at chasedoelling.com.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300