A Guide to REST and API Design

technologies

If the Only Tool You Have is a Hammer...

In his 1966 book “The Psychology of Science,” American psychologist Abraham Maslow tackled the idea that those in the field of psychology needed to approach treatment from multiple perspectives, to take on new ideas, and not just continue using the same theories and techniques created by Freud and his followers so many years ago. Acknowledging that changing your point of view can be difficult, Maslow wrote “[I]t is tempting, if the only tool you have is a hammer, to treat everything like a nail.” We have all had this experience. We get so used to the way things have been done in the past, we sometimes don’t question the reasons for doing them.

It may seem curious to refer to psychology in a work on REST and API Design, but it works to illustrate two distinctive points: (1) that all design decisions, regardless of whether they pertain to software or architecture, should be made within the context of functional, behavioral, and social requirements—not random trends; (2) when you only know how to do one thing well, everything tends to look identical.

“If all you have is a hammer, then everything looks like a nail.”

—Abraham Maslow, The Psychology of Science

In his dissertation, “Architectural Styles and the Design of Network-based Software Architectures,”[1] Roy Fielding defines Representational State Transfer (REST): “Consider how often we see software projects begin with the adoption of the latest fad in architectural design, and only later discover whether or not the system requirements call for such an architecture.”

Don’t have time to read Fielding’s full dissertation? That’s OK. We created this high-level overview with you in mind. To get started, let’s take a look at REST in some detail.

[1] Architectural Styles and the Design of Network-based Software Architectures

Style vs. Standard

An architectural style is an abstraction—not a concrete thing. Take, for example, a Gothic cathedral. The cathedral is different from the Gothic architectural style. The Gothic style defines the attributes or characteristics you would see in a cathedral built in that style.

Comparatively, the National Institute of Standards (NIST) and the National Electrical Codes (NEC) are governing bodies that produce rules we recognize as standards. If you failed to wire a building correctly, the place could burn down to the ground. People often get confused between *standards*—that which we know is right or wrong—and *styles*; a particular mode of expression.

On the Internet, REST is a style and Hypertext Transfer Protocol (HTTP) is a standard. REST relies on stateless, client-server cacheable communications protocols like HTTP to facilitate application development. By applying REST design principles to a protocol, such as HTTP, developers can build interfaces that can be used from nearly any device or operating system.

What's your style?

Three common Web architecture styles are:

- Tunneling (Simple Object Access Protocol—SOAP)
- Objects (Create/Read/Update/Delete—CRUD)
- Hypermedia (Representational State Transfer—REST)

Watch this video[2] to understand the key features of common architectural styles and decide which best fits your needs.

Tutorial: Common Web Architecture Styles

Understand the key features of common architectural styles and decide which best fits your needs

[2] API Design, Lesson 201: Web API Architectural Styles, API Academy.

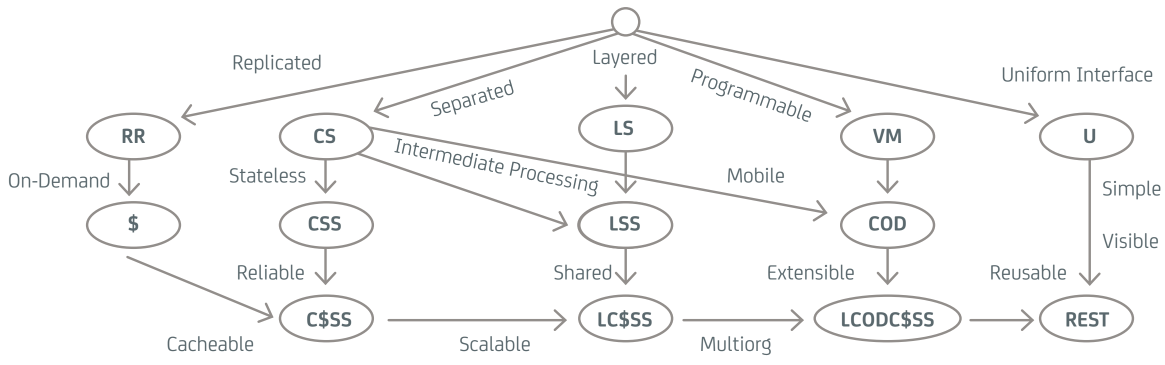
Styles Are Described by Constraints

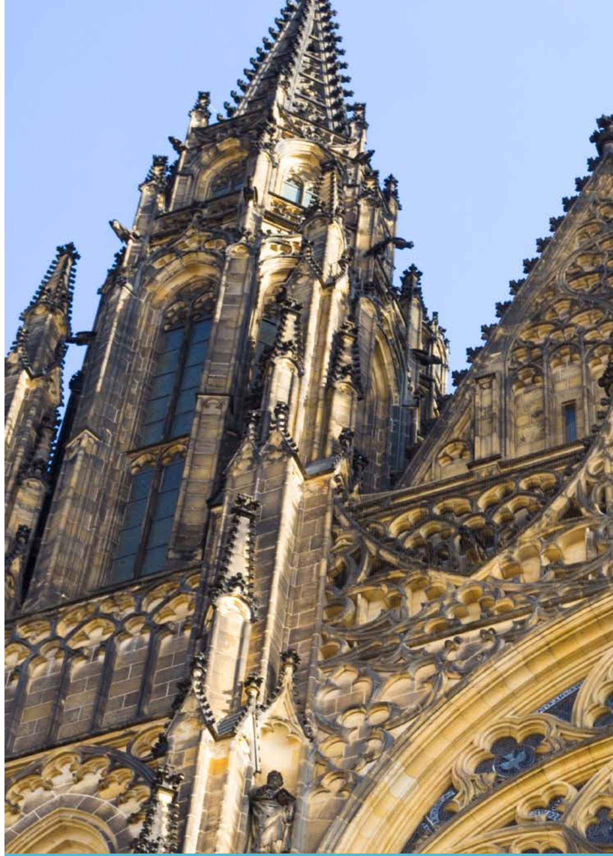
An architectural style is described by the features that make a building or other structure notable and identifiable. The characteristic forms of Gothic architecture, for example, include the pointed arch, the rib vault, buttresses, large windows which are often grouped, rose windows, towers, spires, pinnacles, and ornate facades.

Similarly, REST is described by a set of architectural constraints that attempt to minimize latency and network communications while, at the same time, maximizing the independence and scalability of component implementations. The six constraints of REST include:

1. **Client-Server**—requires that a service offer one or more operations and that services wait for clients to request these operations.
2. **Stateless**—requires communication between service consumer (client) and service provider (server) to be stateless.
3. **Cache**—requires responses to be clearly labeled as cacheable or non-cacheable.
4. **Uniform Interface**—requires all service providers and consumers within a REST-compliant architecture to share a single common interface for all operations.
5. **Layered System**—requires the ability to add or remove intermediaries at runtime without disrupting the system.
6. **Code-on-Demand (optional)**—allows logic within clients (such as Web browsers) to be updated independently from server-side logic using executable code shipped from service providers to consumers.

Figure: REST Derivation by Style Constraints





"Here is one of the few effective keys to the design problem—the ability of the designer to recognize as many of the constraints as possible—his willingness and enthusiasm for working within these constraints. Constraints of price, of size, of strength, of balance, of surface, of time and so forth."[3]

—Charles Eames, Eames Design.

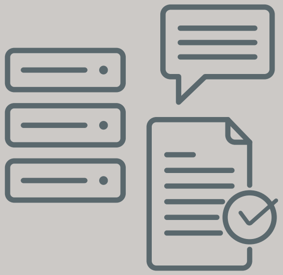
[3] http://www.eamesoffice.com/the-work/design-q-a-text/

Connector ≠ Component

According to Fielding, “[REST] is achieved by placing constraints on connector semantics where other styles have focused on component semantics.” His design focuses on constraining the way things connect to each other—not the way they operate internally—and he applies this theory to the entire network. When building large-scale applications, the concept that the connector is not the same as the component is often overlooked. But Fielding brings it to the forefront.

So, what are components? They include:

- Databases
- File systems
- Message queues
- Transaction manager tools
- Source code



These differ from connectors, which include:

- Web servers
- Browser agents
- Proxy servers
- Shared cache

Components work to solve problems in unique ways. MySQL functions differently from SQL server, as does CouchDB or MongoDB. The same can be said for file systems that are UNIX-, Windows-, or Mac-based. The way you queue up information, the way you decide when a transaction starts and ends; these are entirely local features of the component which can be manipulated by the developer. They are the developer’s components, his/her operating system, tools and language, and are therefore private.

Connectors, on the other hand, are public. They are a series of standardized pipes that all developers work with. Based on Fielding’s principle, developers can be as creative or as mundane as they want within their private components as long as they agree to transmit information back and forth using standardized public connectors.

Keep components and connectors separate, making it easier to interchange them later on. For example, the code you write for your Web server is designed to speak to many devices on the public Internet. But, the code you write for your components is designed to speak specifically to the tools you have available to you.

Ensuring Connectors Work Together

When building client-based applications or server-side services, it is this matching of private components to public connectors—wiring them up and chaining them together—that can make development both challenging and exciting. So, how do you ensure a connector works? How can you design scalable applications if they're communicating over connectors?

Identification of Resources— URIs, URLs, and URNs as identifiers

Resource Representations— media types as ways to represent information passed between parties

Self-Describing Messages— combining metadata in headers, as well as the body of a message, to create a self-descriptive response

Hypermedia— links and forms as a way to describe to the client the available actions currently supported by the service

The uniform interface constraint is fundamental to the design of any REST service. It simplifies and decouples the connectors, which enables each part to evolve independently. Because of how the Web is used today, the four constraints above are the essential tools that help developers realize Fielding's uniform interface. The next few pages explain these tools in greater depth.

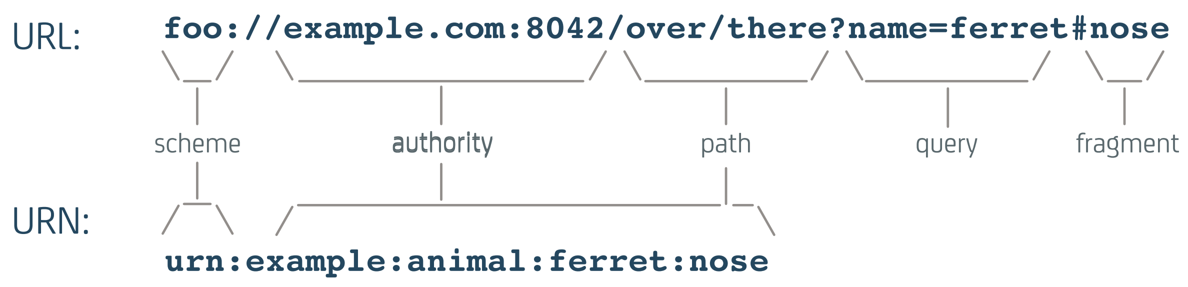
URIs for Identification

As RFC2396[4] describes it, “a Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource.” This identifier can be realized in one of two ways, a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). URLs (e.g. http://example.org/users/mike) are used to identify the online location of an individual resource while URNs (e.g. urn:user:mike) are intended to be persistent, location-independent identifiers. The URN functions like a person’s name and a URL resembles that person’s street address. In other words, the URN defines an item’s identity (“the user’s name is mike”) and the URL provides a method for finding it (“mike can be found at example.org/users/”).

The components of a URI include:

- **Scheme Name**—identifies the protocol (e.g., FTP:, HTTP:, HTTPS:, IRC:)
- **Hierarchical Part**—intended to hold information hierarchical in nature
    - **Authority**—refers to the actual DNS resolution of the server (e.g., domain name or IP address)
    - **Path**—pertains to a sequence of segments separated by a forward slash (“/”)
- **Query**—contains additional identification information that is non-hierarchical in nature and often separated by a question mark (“?”)
- **Fragment**—provides direction to a secondary resource within the primary one identified by the Authority and Path and separated from the rest by a hash (“#”)

The structure of URIs


URL:
foo://example.com:8042/over/there?name=ferret#nose
scheme
authority
path
query
fragment
URN:
urn:example:animal:ferret:nose

[4] https://tools.ietf.org/html/rfc2396

Media Types for Representation

According to RFC2046[5], MIME type identifiers (media types) should be used to "specify the nature of the data in the body of a MIME entity, along with any auxiliary information that may be required." MIME types were first used for email transmissions, as is evidenced by its full name: Multipurpose Internet Mail Extensions. Today, MIME types permit people to exchange different kinds of data files on the Internet: audio, video, text, images and application programs.

MIME types (or media types) identify the nature of the data and auxiliary information. On the Web, media types also identify processing rules for the message. The MIME type identifier string has a type and subtype separated by a slash (e.g., text/plain, image/gif, etc.).

In addition to standard MIME type strings (e.g. application/json), identifiers can be created using the following conventions:

- Use vnd. as a prefix to the subtype for vendor-specific MIME types which are part of a commercial product (e.g. vnd.bigcompany.report/json).
- Use prs. as a prefix to the subtype for personal/vanity MIME types which are not part of a commercial product (e.g. prs.smith.data/json).

[5] https://tools.ietf.org/html/rfc2396

MIME Type Registry

A complete list of official MIME types assigned by the Internet Assigned Number Authority (IANA) can be found here.

Headers+Body for Self-Describing Messages

RFC2616[6] states that [in HTTP] messages consist of a start-line, zero or more header fields (also known as “header”), an empty line (e.g., a line with nothing preceding the CRFL) indicating the end of the header fields, and possibly a message-body.

Each client request and server response is a message, and REST-compliant applications expect each message to be self-descriptive. That means each message must contain all the information necessary to complete the task. Other ways to describe this kind of message are “stateless” or “context-free.” Each message passed between client and server can have a body and metadata.

REST implementations also depend on the notion of a constrained set of operations that are fully understood by both client and server at the outset. In HTTP, the operations are described on the “start line” and the six most widely used operations in HTTP are:

- **GET**—return whatever information is identified by the Request-URI
- **HEAD**—identical to GET except that the server must not return a message-body in the response, only the metadata
- **OPTIONS**—return information about the communication options available on the request/response chain identified by the Request-URI
- **PUT**—requests that the enclosed entity be stored under the supplied Request-URI
- **POST**—requests that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI
- **DELETE**—requests that the origin server delete the resource identified by the Request-URI

The first three are read-only operations, while the last three are write operations. In HTTP, there are well-defined rules for how clients and servers are expected to behave when using these operators. The names and meanings of the accompanying metadata elements (headers) are also well-defined. REST-compliant applications running over HTTP understand and follow these rules very carefully.

[6] https://tools.ietf.org/html/rfc2396

Sample HTTP “GET” Exchange

To retrieve a file at http://www.somehost.com/path/file.html, open a socket to the host www.somehost.com, use the default port of 80 because none is specified in the URL, and send the following through the socket:

```
GET/path/file.html HTTP/1.0
From: someuser@jmarshall.com
User-Agent: HTTPTool/1.0
[blank line here]
```

Sent back through the same socket, the server should respond with:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Year!</h1>
(more file contents)
</body>
</html>
```

Links and Forms for Hypermedia

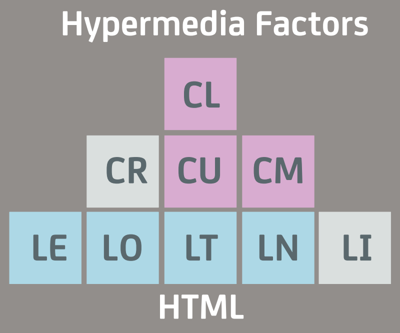
Links and forms are used within a media type to support Fielding's hypermedia constraint. For example, there are a handful of affordances in HTML and the common Web browser understands the rules for all of them. The links and forms in an HTML message are easy to recognize, but what might not be so clear are the processing rules, or the semantics, that are associated with them.

Links and forms afford you the ability to take an action—they are *affordances*. HTML has a well-defined set of affordances. Some affordances allow you to write data, like a form that has the method property set to "POST." Some affordances allow you to pull data from a remote location to view within the current HTML document like the IMG element. The HTML A element is an affordance for navigating to a new location on the Web.

According to Fielding, "Hypermedia is defined by the presence of application control information embedded within, or as a layer above, the presentation of information." For Fielding, REST offers service providers the ability to send control information (links and forms) to client applications across the world by sending affordances, the hypermedia controls.

H Factors

When comparing media types, it can be helpful to document the existing H Factors in a simple visual chart. In the example below, the bottom row identifies basic link factors—the most noticeable hypermedia factors—while the top two rows identify control data factors.

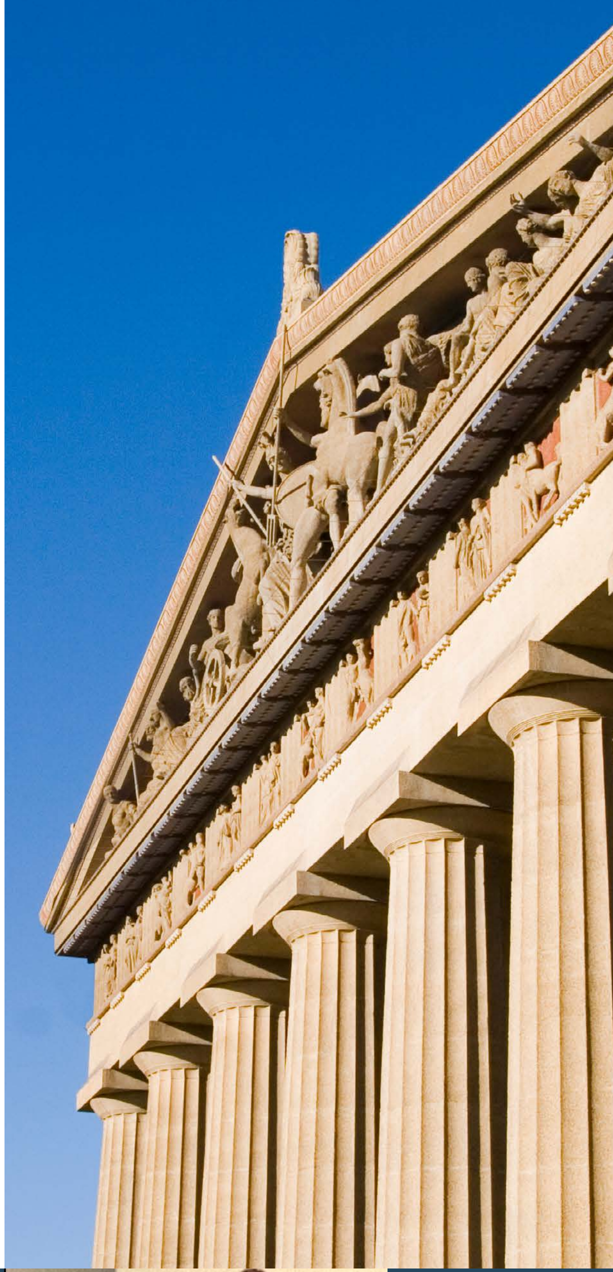


Link Support
[LE] Embedding links
[LO] Outbound links
[LT] Templated queries
[LN] Non-Idempotent updates
[LI] Idempotent updates

Control Data Support
[CR] Control data for read requests
[CU] Control data for update requests
[CM] Control data for interface methods
[CL] Control data for links

For more information on H-Factors:
http://amundsen.com/hypermedia/hfactor/.

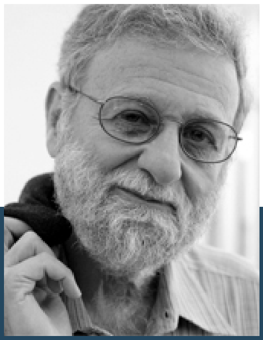
Software That Spans Lifetimes



Quality physical architecture spans lifetimes. Buildings that were constructed hundreds of years ago can be just as lively, just as useful, just as vibrant and comforting today as they were when people first stepped inside—even when these building are converted from their original uses, such as when churches are transformed into museums or meeting halls into apartment buildings. Why? Because these timeless buildings rely on universal architectural patterns that cross space and time. An entryway is an entryway; a window a window. How these elements are implemented is dependent on the available materials. How they are represented is different in each local case, but they can still be identified year over year.

In software development, the concept is the same. Sometimes software architects and developers want to build applications that can last a long time. REST, with its set of universal constraints (like architectural patterns), is one way of accomplishing this.

But Fielding didn't say this is the only way to be successful. When he was writing his dissertation, he didn't include all the possibilities, all the answers. In fact, there were many parts left unfinished. What Fielding did do, however, was document his approach to creating an architectural style for networked software which was based on a identifying a series of constraints to meet his goal of minimizing latency and maximizing scalability. In fact, he used constraints in the same way that Charles Eames used constraints—because they helped achieve his goal.



We can take advantage of Fielding's experience in order to help us see things a little differently. We can use the tools he provided to experiment with constraints and express our own style—and, in the process, build remarkable software applications that can, if we wish, stand the test of time.

"The value of a well-designed object is when it has such a rich set of affordances that the people who use it can do things with it that the designer never imagined."[7]

—Donald Norman, 1994

[7] https://www.youtube.com/watch?v=NK1Zb_5VxuM

Learn more about CA API Management

Visit **ca.com/API**

CA Technologies (NASDAQ: CA) creates software that fuels transformation for companies and enables them to seize the opportunities of the application economy. Software is at the heart of every business, in every industry. From planning to development to management and security, CA is working with companies worldwide to change the way we live, transact, and communicate – across mobile, private, and public cloud, distributed and mainframe environments. Learn more at ca.com.

CS200-110010

ca® technologies