**Chapter 1**

**Fundamentals of RESTful APIs**

Sanjay Patnia\*

a UCSC Extension, Santa Clara, California, USA

APIs are not new. They’ve served as interfaces that enable applications to communicate with each other for decades. But the role of APIs has changed dramatically in the last few years. Innovative companies have discovered that APIs can be used as an interface to the business, allowing them to monetize digital assets, extend their value proposition with partner-delivered capabilities, and connect to customers across channels and devices. When you create an API, you are allowing others within or outside of your organization to make use of your service or product to create new applications, attract customers, or expand their business. Internal APIs enhance the productivity of development teams by maximizing reusability and enforcing consistency in new applications. Public APIs can add value to your business by allowing third party developers to enhance your services or bring their customers to you. As developers find new applications for your services and data, a network effect occurs, delivering significant bottom-line business impact. For example, Expedia opened up their travel booking services to partners through an API to launch the Expedia Affiliate Network, building a new revenue stream that now contributes $2B in annual revenue. Salesforce released APIs to enable partners to extend the capabilities of their platform and now generates half of their annual revenue through those APIs, which could be SOAP-based (JAX-WS) and, more recently, RESTful (JAX-RS), Spring Boot and now Micronaut.

SOAP web service depends upon a number of technologies (such as UDDI, WSDL, SOAP, HTTP) and protocols to transport and transform data between a service provider and the consumer, and can be created with JAX-WS.

Later, Roy Fielding (in the year 2000) presented his doctoral dissertation, “Architectural Styles and the Design of Network-based Software Architecture.” He coined the term “REST,” an architectural style for distributed hypermedia systems. Put simply, REST (short for REpresentational State Transfer) is an architectural style defined to help create and organize distributed systems. The key word from that definition should be “style,” because an important aspect of REST (and which is one of the main reasons books like this one exist) is that it is an architectural style—not a guideline, not a standard, or anything that would imply that there are a set of hard rules to follow in order to end up having a RESTful architecture.

In this chapter, I’ll be covering REST fundamentals, SOAP vs. REST, and Web Architectural Style to provide a solid foundation and better prepare you for what you’ll see in later chapters.

# REST

The main idea behind REST is that a distributed system, organized RESTfully, will improve in the following areas:

* **Performance**: The communication style proposed by REST is meant to be efficient and simple, allowing a performance boost on systems that adopt it.
* **Scalability of component** **interaction**: Any distributed system should be able to handle this aspect well enough, and the simple interaction proposed by REST greatly allows for this.
* **Simplicity of interface**: A simple interface allows for simpler interactions between systems, which in turn can grant benefits like the ones previously mentioned.
* **Modifiability of** **components**: The distributed nature of the system, and the separation of concerns proposed by REST (more on this in a bit), allows for components to be modified independently of each other at a minimum cost and risk.
* **Portability**: REST is technology- and language-agnostic, meaning that it can be implemented and consumed by any type of technology (there are some constraints that I’ll go over in a bit, but no specific technology is enforced).
* **Reliability**: The stateless constraint proposed by REST (more on this later) allows for the easier recovery of a system after failure.
* **Visibility**: Again, the stateless constraint proposed has the added full state of said request (this will become clear once I talk about the constraints in a bit). From this list, some direct benefits can be extrapolated. A component-centric design allows you to make systems that are very fault-tolerant. Having the failure of one component not affect the entire stability of the system is a great benefit for any system. Interconnecting components is quite easy, minimizing the risks when adding new features or scaling up or down. A system designed with REST in mind will be accessible to a wider audience, thanks to its portability (as described earlier). With a generic interface, the system can be used by a wider range of developers. In order to achieve these properties and benefits, a set of constraints were added to REST to help define a uniform connector interface. REST is not suggested to use when you need to enforce a strict contract between client and server and when performing transactions that involve multiple calls.

**SOAP vs. REST**

Table [1-1](#bookmark) has a comparison between SOAP and REST with an example of use cases each can support.

***Table 1-1.*** *SOAP vs. REST comparison*

|  |  |  |
| --- | --- | --- |
| Topic | SOAP | REST |
| Origin | SOAP (Simple Object Access Protocol) was created in 1998 by Dave Winer et al. in collaboration with Microsoft. Developed by a large software company, this protocol addresses the goal of addressing the needs of the enterprise market. | REST (Representational State Transfer) was created in 2000 by Roy Fielding at UC, Irvine. Developed in an academic environment, this protocol embraces the philosophy of the open Web. |
| Basic Concept | Makes data available as services (verb + noun), for example “getUser” or “PayInvoice” | Makes data available as resources (nouns), for example “user” or “invoice” |
| Pros | Follows a formal enterprise approach  Works on top of any communication protocol, even asynchronously  Information about objects is communicated to clients.  Security and authorization are part of the protocol.  Can be fully described using WSDL | Follows the philosophy of the Open Web  Relatively easy to implement and maintain  Clearly separates client and server implementations  Communication isn’t controlled by a single entity  Information can be stored by the client to prevent multiple calls.  Can return data in multiple formats (JSON, XML etc.) |
| Cons | Spends a lot of bandwidth communicating metadata  Hard to implement and is unpopular among Web and mobile developers | Only works on top of the HTTP protocol  Hard to enforce authorization and security on top of it |
| When to use | When clients need to have access to objects available on servers  When you want to enforce a formal contract between client and server | When clients and servers operate on a Web environment  When information about objects doesn’t need to be communicated to the client |
| When not to use | When you want the majority of developers to easily use your API  When your bandwidth is very limited | When you need to enforce a strict contract between client and server  When performing transactions that involve multiple calls |
| Use cases | Financial services  Payment gateways  Telecommunication services | Social media services  Social networks  Web chat services  Mobile services |
| Examples | <https://www.salesforce.com/developer/docs/api/> - Salesforce SOAP API  <https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture/> -Paypal SOAP API | <https://dev.twitter.com/>  <https://developer.linkedin.com/apis> |
| Conclusion | Use SOAP if you are dealing with transactional operations and you already have an audience that is satisfied with this technology. | Use REST if you’re focused on wide-scale API adoption or if your API is targeted at mobile apps. |

**Web Architectural Style**

According to Fielding, there are two ways to define a system.

* One is to start from a blank slate—an empty whiteboard—with no initial knowledge of the system being built or the use of familiar components until the needs are satisfied.
* A second approach is to start with the full set of needs for the system, and constraints are added to individual components until the forces that influence the system are able to interact in harmony with each other.

REST follows the second approach. In order to define a REST architecture, a null-state is initially defined—a system that has no constraints whatsoever and where component differentiation is nothing but a myth—and constraints are added one by one. The following subsections cover web architectural style constraints. Each of these constrains defines how the framework for REST APIs should be architected and designed. Security is another aspect which needs to be considered independently as part of this framework when rolling out RESTful APIs to the end users.

**Client-Server**

The separation of concerns is the core theme of the Web’s client-server constraints.

The Web is a client-server-based system, in which clients and servers have distinct parts to play.

They may be implemented and deployed independently, using any language or technology, so long as they conform to the Web’s uniform interface.

**Uniform Resource Interface**

The interactions between the Web’s components—meaning its clients, servers, and network-based intermediaries—depend on the uniformity of their interfaces.

Web components interoperate consistently within the uniform interface’s four constraints, which Fielding identified as:

* Identification of resources
* Manipulation of resources through representations
* Self-descriptive messages
* Hypermedia as the engine of application state (HATEOAS)

**Layered System**

Generally speaking, a network-based intermediary will intercept client-server communication for a specific purpose.

Network-based intermediaries are commonly used for enforcement of security, response caching, and load balancing

The layered system constraints enable network-based intermediaries such as proxies and gateways to be transparently deployed between a client and server using the Web’s uniform interface.

**Caching**

Caching is one of web architecture’s most important constraints. The cache constraints instruct a web server to declare the cache ability of each response’s data.

Caching response data can help to reduce client-perceived latency, increase the overall availability and reliability of an application, and control a web server’s load. In a word, caching reduces the overall cost of the Web.

**Stateless**

The stateless constraint dictates that a web server is not required to memorize the state of its client applications. As a result, each client must include all of the contextual information that it considers relevant in each interaction with the web server.

Web servers ask clients to manage the complexity of communicating their application state so that the web server can service a much larger number of clients. This trade-off is a key contributor to the scalability of the Web’s architectural style.

**Code-on-Demand**

The Web makes heavy use of code-on-demand, a constraint which enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients.

Code-on-demand tends to establish a technology coupling between web servers and their clients, since the client must be able to understand and execute the code that it downloads on-demand from the server. For this reason, code-on-demand is the only constraint of the Web’s architectural style that is considered optional.

**HATEOAS**

The final principle of REST is the idea of using Hypermedia As The Engine Of Application State (HATEOAS). When developing a client-server solution using HATEOAS, the logic on the server side might change independently of the clients.

Hypermedia is a document-centric approach with the added support for embedding links to other services and information within the document format.

One of the uses of hypermedia and hyperlinks is composing complex sets of information from disparate sources. The information could be within a company private cloud or within a public cloud from disparate sources.

Example:

<podcast id="111">

<customer>http://customers.myintranet.com/customers/1</customers>

<link>http://podcast.com/myfirstpodcast</link>

<description> This is my first podcast </description>

</podcast>

Each of these web architecture styles adds beneficial properties to the web system.

By adopting these constraints, teams can build simple, visible, usable, accessible, evolvable, flexible, maintainable, reliable, scalable and performant systems as shown in Table [1-2](#bookmark1) below:

**Table 1-2.** Constraint and system property

|  |  |
| --- | --- |
| By following the constraint | Gain the following system property |
| Client-server interactions | Simple, Evolvable, Scalable |
| Stateless communications | Simple, Visible, Maintainable, Evolvable, and Reliable |
| Cacheable data | Visible, Scalable, and Performant |
| Uniform Interfaces | Simple, Usable, Visible, Accessible, Evolvable, and Reliable |
| Layered system | Flexible, Scalable, Reliable, and Per formant |
| Code on demand | Evolvable |

Note: I have not covered security in this chapter as part of REST fundamentals, but security is very important for rolling out RESTful APIs.

**What is REST?**

We have briefly introduced REST with REST API fundamentals in the previous section. This section has further introductory details about REST concepts.

“REST” was coined by Roy Fielding in his Ph.D. dissertation to describe a design pattern for implementing networked systems. REST is Representational State Transfer, an architectural style for designing distributed systems. It’s not a standard, but rather a set of constraints. It’s not tied to HTTP, but is associated most commonly with it.

**REST Basics**

Unlike SOAP and XML-RPC, REST does not really require a new message format. The HTTP API is CRUD (Create, Retrieve, Update, and Delete)

* GET = “give me some info” (Retrieve)
* POST = “here’s some update info” (Update)
* PUT = “here’s some new info” (Create)
* DELETE = “delete some info” (Delete)
* And more….
* PATCH = The HTTP method PATCH can be used to update partial resources. For instance, when you only need to update one field of the resource, PUTting a complete resource representation might be cumbersome and utilizes more bandwidth.
* HEAD = The **HEAD** method is identical to the GET method, except that the server must not return a message body in the response. This method is often used for testing hypertext links for validity, accessibility, and recent modification.
* OPTIONS = This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.
* Notion of “Idempotency” – the idea that when sending a GET, DELETE, or PUT to the system, the effect should be the same whether the command is sent one or more times, but POST creates an entity in the collection and therefore is not idempotent.

**REST Fundamentals**

Just to remind you, about 8,356 APIs were written in REST by ProgrammableWeb.com in 2016. REST is resource-based architecture. A resource is accessed via a common interface based on the HTTP standard methods. REST asks developers to use HTTP methods explicitly and in a way that’s consistent with the protocol definition. Each resource is identified by a URL. Every resource should support the HTTP common operations, and REST allows that resource to have different representations, e.g., text, xml, json, etc. The rest client can ask for specific representation via the HTTP protocol (Content Negotiation). Table [1-3](#bookmark2) below describes data elements used in REST.

**Table 1-3.** Structures of REST

|  |  |
| --- | --- |
| Data Element | Description |
| Resource | Conceptual target of a hypertext reference, e.g., customer/order |
| Resource Identifier | A uniform resource locator (URL) or uniform resource name (URN) identifying a specific resource, e.g., <http://myrest.com/customer/3435> |
| Resource Metadata | Information describing the resource, e.g., tag, author, source link, alternate location, alias names |
| Representation | The resource content—JSON Message, HTML Document, JPEG Image |
| Representation Metadata | Information describing how to process the representation, e.g., media type, last-modified time |
| Control Data | Information describing how to optimize response processing, e.g., if-modified-since, cache-control-expiry |

Let’s look at some examples.

**Resources**

First, a REST resource to GET a list of podcasts:

http://prorest/podcasts

Next, a REST resource to GET details of podcast id 1:

http://prorest/podcasts/1

**Representations**

Here is an XML representation of a response—GET customer for an id.

<Customer>

<id>123</id>

<name>John</name>

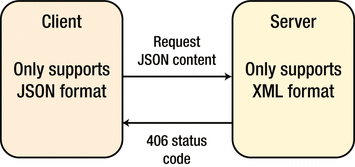
</Customer>

Next, a JSON representation of a response—GET customer for an id:

{"Customer":{"id":"123","name":"John"}}

**Content Negotiation**

HTTP natively supports a mechanism based on headers to tell the server about the content you expect and you’re able to handle. Based on these hints, the server is responsible for returning the corresponding content in the correct format. Figure [1-1](#bookmark3) shows an example.



***Figure 1-1.*** *Content negotiation*

If the server doesn’t support the requested format, it will send back a 406 status code (Not Acceptable) to notify the client that made the request (“The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request”) according to the specification.

**To Review**

REST identifies the key architectural principles of why the Web is prevalent and scalable. The next step in the education of the Web is to apply these principles to the semantics Web and the world of web services. REST offers a simple, interoperable, and flexible way of writing web services that can be very different than the WS-\* that so many of you had training in. In the next chapter we will introduce **Micronaut** - A MODERN, JVM-BASED, FULL-STACK FRAMEWORK FOR BUILDING MODULAR, EASILY TESTABLE MICROSERVICE AND SERVERLESS APPLICATIONS. We will also compare it with similar framework Spring Boot.