What is a REST API?

REST or RESTful API (https://www.mulesoft.com/resources/api/restful-api) design (Representational State Transfer) is designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it usually takes advantage of HTTP when used for Web APIs. This means that developers do not need to install libraries or additional software in order to take advantage of a REST API design. REST API Design was defined by Dr. Roy Fielding in his 2000 doctorate dissertation. It is notable for its incredible layer of flexibility. Since data is not tied to methods and resources, REST has the ability to handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

This freedom and flexibility inherent in REST API design allow you to build an API that meets your needs while also meeting the needs of very diverse customers. Unlike SOAP (https://www.mulesoft.com/webinars/api/soap-connect), REST is not constrained to XML, but instead can return XML, JSON, YAML or any other format depending on what the client requests. And unlike RPC, users aren't required to know procedure names or specific parameters in a specific order.

However, there are drawbacks to REST API design. You can lose the ability to maintain state in REST, such as within sessions, and it can be more difficult for newer developers to use. It's also important to understand what makes a REST API RESTful, and why these constraints exist before building your API (https://www.mulesoft.com/lp/ebook/api/building-api-blueprint). After all, if you do not understand why something is designed in the manner it is, you can hinder your efforts without even realizing it.

Understanding REST API Design

While most APIs claim to be RESTful, they fall short of the requirements and constraints asserted by Dr. Fielding. There are six key constraints to REST API design to be aware of when deciding whether this is the right API type (https://www.mulesoft.com/resources/api/types-of-apis) for your project.

Client-Server

The client-server constraint works on the concept that the client and the server should be separate from each other and allowed to evolve individually and independently. In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client. This creates a separation of concerns, letting each application grow and scale independently of the other and allowing your organization to grow quickly and efficiently.

Stateless

REST APIs are stateless, meaning that calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully. A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself. Identifying information is not being stored on the server when making calls. Instead, each call has the necessary data in itself, such as the API key, access token, user ID, etc. This also helps increase the API's reliability by having all of the data necessary to make the call, instead of relying on a series of calls with server state to create an object, which may result in partial fails. Instead, in order to reduce memory requirements and keep your application as scalable as possible, a RESTful API requires that any state is stored on the client—not on the server.

Cache

Because a stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data. This means that when data is cacheable, the response should indicate that the data can be stored up to a certain time (expires-at), or in cases where data needs to be real-time, that the response should not be cached by the client. By enabling this critical constraint, you will not only greatly reduce the number of interactions with your API, reducing internal server usage, but also provide your API users with the tools necessary to provide the fastest and most efficient apps possible. Keep in mind that caching is done on the client side. While you may be able to cache some data within your architecture to perform overall performance, the intent is to instruct the client on how it should proceed and whether or not the client can store the data temporarily.

Uniform Interface

The key to the decoupling client from server is having a uniform interface that allows independent evolution of the application without having the application's services, models, or actions tightly coupled to the API layer (https://www.mulesoft.com/resources/api/api-layer) itself. The uniform interface lets the client talk to the server in a single language, independent of the architectural backend of either. This interface should provide an unchanging, standardized means of communicating between the client and the server, such as using HTTP with URI resources, CRUD (Create, Read, Update, Delete), and JSON.

Layered System

As the name implies, a layered system is a system comprised of layers, with each layer having a specific functionality and responsibility. If we think of a Model View Controller framework, each layer has its own responsibilities, with the models comprising how the data should be formed, the controller focusing on the incoming actions and the view focusing on the output. Each layer is separate but also interacts with the other. In REST API design, the same principle holds true, with different layers of the architecture working together to build a hierarchy that helps create a more scalable and modular application.

A layered system also lets you encapsulate legacy systems and move less commonly accessed functionality to a shared intermediary while also shielding more modern and commonly used components from them. Additionally, the layered system gives you the freedom to move systems in and out of your architecture as technologies and services evolve, increasing flexibility and longevity as long as you keep the different modules as loosely coupled as possible. There are substantial security benefits (https://www.mulesoft.com/webinars/api/security-best-practices) of having a layered system since it allows you to stop attacks at the proxy layer, or within other layers, preventing them from getting to your actual server architecture. By utilizing a layered system with a proxy, or creating a single point of access, you are able to keep critical and more vulnerable aspects of your architecture behind a firewall, preventing direct interaction with them by the client. Keep in mind that security is not based on single “stop all” solution, but rather on having multiple layers with the understanding that certain security checks may fail or be bypassed. As such, the more security you are able to implement into your system, the more likely you are to prevent damaging Attacks.

Code on Demand

Perhaps the least known of the six constraints, and the only optional constraint, Code on Demand allows for code or applets to be transmitted via the API for use within the application. In essence, it creates a smart application that is no longer solely dependent on its own code structure. However, perhaps because it’s ahead of its time, Code on Demand has struggled for adoption as Web APIs are consumed across multiple languages and the transmission of code raises security questions and concerns. (For example, the directory would have to be writeable, and the firewall would have to let what may normally be restricted content through.)

Together, these constraints make up the theory of Representational State Transfer (https://www.mulesoft.com/resources/esb/restful-web-services-esb-rest-services-integration), or REST. As you look back through these you can see how each successive constraint builds on top of the previous, eventually creating a rather complex—but powerful and flexible—application program interface. But most importantly, these constraints make up a design that operates similarly to how we access pages in our browsers on the World Wide Web. It creates an API that is not dictated by its architecture, but by the representations that it returns, and an API that—while architecturally stateless—relies on the representation to dictate the application’s state.

For more information about REST API Design (https://www.mulesoft.com/platform/api/anypoint-designer), check out the eBook Undisturbed REST: A Guide to Designing the Perfect API (https://www.mulesoft.com/lp/ebook/api/restbook).

Related resources

The value of APIs for business (https://www.mulesoft.com/resources/api/connected-business-strategy)

What is REST API design? (https://www.mulesoft.com/resources/api/what-is-rest-api-design)

API development best practices (https://www.mulesoft.com/resources/api/development-best-practices)

Recommended for you

Connectivity benchmark report (https://www.mulesoft.com/lp/reports/connectivity-benchmark)

The application network (https://www.mulesoft.com/lp/whitepaper/api/application-network)

How to design and manage APIs (https://www.mulesoft.com/lp/whitepaper/api/design-apis)

Watch now on demand

Best practices for microservices (https://www.mulesoft.com/webinars/api/microservices-architecture)

API security best practices (https://www.mulesoft.com/webinars/api/security-best-practices)

Anypoint Platform overview (https://www.mulesoft.com/webinars/api/mule-101-anypoint-platform-overview)