REST VS CRUD: EXPLAINING REST & CRUD OPERATIONS

Some of the confusion around REST and CRUD has to do with the overlapping of basic commands mandated by both processes. This is further amplified by the Rails community embracing REST and its *GET, PUT, POST* nature.

Savvy developers can see glaring similarities between *GET, PUT, POST* and *CREATE, READ, UPDATE, DELETE*. The latter commands are the foundation of CRUD. And while the similarities cannot be ignored, it should be noted that REST is not simply a carbon copy of CRUD.

REST: Foundation & Principles

Each REST command is centered around a resource. In REST, a resource is really anything that can be pointed to via HTTP protocol. For example, an image, a website, a document, or a weather service. The possibilities are almost endless.

In plain terms, REST stands for Representational State Transfer, an architectural style designed for distributed hypermedia, or an Application Programming Interface. You've probably heard the latter referred to as an API. Another way to think of an API is to define it as a web service that conforms to the architectural principles of REST. Each API is called by issuing a standard HTTP request method: POST, GET, PUT, and less commonly, DELETE. DELETE is usually implied, though not necessarily stated.

The terms that define REST principles were introduced in Dr. Roy Fieldings' thesis, "Architectural

Styles and the Design of Network-Based Software Infrastructure." Overall, REST can be thought of as the standard in service application development. It offers an alternative to:

- Simple Object Access Protocol (SOAP)
- Common Object Request Broker Architecture (CORBA)
- RMI
- Many others

Principles of REST

There are six guiding constraints of REST:

- Client-server mandata
- Statelessness
- Cache
- Interface/uniform contract
- Layered system
- Code-on-demand (optional)

Let's look at each.

Client-server mandate

This mandate underscores the fact that REST is a distributed approach via the nature of separation between client and server. Each service has multiple capabilities and listens for requests. Requests are made by a consumer and accepted or rejected by the server.

Statelessness

Due to the nature of statelessness, it is a guiding principle of RESTful architecture. It mandates what kind of commands can be offered between client and server. Implementing stateless requests means the communication between consumer and service is initiated by the request, and the request contains all the information necessary for the server to respond.

Cache

Cache mandates that server responses be labeled as either cacheable or not. Caching helps to mitigate some of the constraints of statelessness. For example, a request that is cached by the consumer in an attempt to avoid re-submitting the same request twice.

Interface/uniform contract

RESTful architecture follows the principles that define a Uniform Contract. This prohibits the use of multiple, self-contained interfaces within an API. Instead, one interface is distributed by hypermedia connections.

Layered system

This principle is the one that makes RESTful architecture so scalable. In a Layered System, multiple

layers are used to grow and expand the interface. None of the layers can see into the other.

This allows for new commands and middleware to be added without impacting the original commands and functioning between client and server.

Optional: Code-on-demand

RESTful applications don't have to include Code-On-Demand, but they must have Client-Server, Statelessness, Caching, Uniform Contract, and Layered Systems. Code-on-Demand allows logic within clients to be separate from that within servers. This allows them to be updated independently of server logic.

REST in a nutshell

REST refers to a set of defining principles for developing APIs. It uses HTTP protocols like GET, PUT, POST to link resources to actions within a client-server relationship. In addition to the client-server mandate, it has several other defining constraints. The principles of RESTful architecture serve to create a stable and reliable application that offers simplicity and end-user satisfaction.

CRUD: Foundation & Principles

With a better understanding of RESTful architecture, it's time to dive into CRUD. CRUD is an acronym for:

- CREATE
- READ
- UPDATE
- DELETE

These form the standard database commands that are the foundation of CRUD. Many software developers view these commands as primitive guidance, at best. That's because CRUD was not developed as a modern way to create API. In fact, CRUD's origins are in database records.

By definition, CRUD is more of a cycle than an architectural system. On any dynamic website, there are likely multiple CRUD cycles that exist. For instance, a buyer on an eCommerce site can CREATE an account, UPDATE account information, and DELETE things from a shopping cart.

A Warehouse Operations Manager using the same site can CREATE shipping records, RETRIEVE them as needed, and UPDATE supply lists. Retrieve is sometimes substituted for READ in the CRUD cycle.

Database Origins

The CRUD cycle is designed as a method of functions for enhancing persistent storage—with a database of records, for instance. As the name suggests, persistent storage outlives the processes that created it. These functions embody all the hallmarks of a relational database application.

In modern software development, CRUD has transcended its origins as foundational functions of a database and now maps itself to design principles for dynamic applications like HTTP protocol, DDS, and SQL.

Principles of CRUD

As mentioned above, the principles of the CRUD cycle are defined as CREATE, READ/RETRIEVE, UPDATE, and DELETE:

- **CREATE** procedures generate new records via INSERT statements.
- **READ** procedures reads the data based on input parameters. Similarly, **RETRIEVE** procedures grab records based on input parameters.
- **UPDATE** procedures modify records without overwriting them.
- **DELETE** procedures delete where specified.

REST & CRUD similarities

If you look at the two as we have described above, it may be difficult to understand why they are often treated in the same way:

- REST is a robust API architecture.
- CRUD is a cycle for keeping records current and permanent.

The lack of clarity between the two is lost for many when they fail to determine when CRUD ends and REST begins. We mentioned above that CRUD can be mapped to DDS, SQL, and HTTP protocols. And that HTTP protocols are the link between resources in RESTful architecture, a core piece of REST's foundation.

Mapping CRUD principles to REST means understanding that GET, PUT, POST and CREATE, READ, UPDATE, DELETE have striking similarities because the former grouping applies the principles of the latter.

However, it is also important to note that a RESTful piece of software architecture means more than mapping GET, PUT, POST commands.

REST vs CRUD: what's the difference?

CRUD is a cycle that can be mapped to REST, by design. Permanence, as defined in the context of CRUD, is a smart way for applications to mitigate operational commands between clients and services.

But REST governs much more than permanence within its principles of architecture. Here are some of the ways that REST is not only different than CRUD but also offers much more:

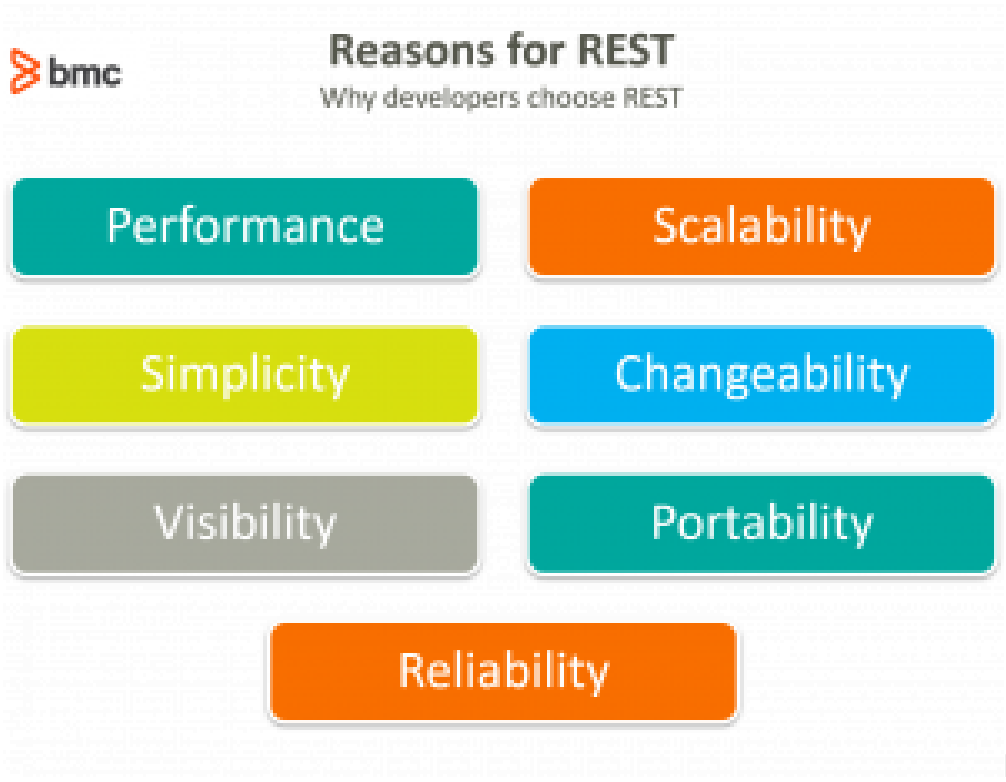
- REST is an architectural system centered around resources and hypermedia, via HTTP protocols
- CRUD is a cycle meant for maintaining permanent records in a database setting
- CRUD principles are mapped to REST commands to comply with the goals of RESTful architecture

In REST:

- Representations must be uniform with regard to resources
- Hypermedia represents relationships between resources
- Only one entry into an API to create one self-contained interface, then hyperlink to create

relationships

The way forward



Not surprisingly, the demand for RESTful architecture has continued to grow over the years. And, as it becomes an even more popular architectural style, developers should be able to make the distinction between the principles of other options like SOAP, CORBA, and RMI.

Additional resources

- BMC DevOps Blog
- BMC IT Operations Blog
- Data Storage Explained: Data Lake vs Warehouse vs Database
- Introduction to Database Reliability
- CAP Theorem for Databases: Consistency, Availability & Partition Tolerance
- How To Create Great Developer APIs