Just because you can interact with a system using HTTP, and send JSON back and forth, doesn’t mean it’s a RESTful system.

I’ll do a quick review of the options developers had back then to interconnect systems, mainly going

over SOAP and XML-RPC (the two main players before REST).

This whole thing started with Roy Fielding, an American computer scientist born in 1965. He is one of the

Main authors of the HTTP protocol (the protocol that the entire Web infrastructure is based on).

In the year 2000, Fielding presented his doctoral dissertation, *Architectural Styles and the Design of*

*Network-based Software Architecture.* In it 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.

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 benefit of

improving visibility, because any monitoring system doesn’t need to look further

than a single request message to determine the 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 Constraints

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.

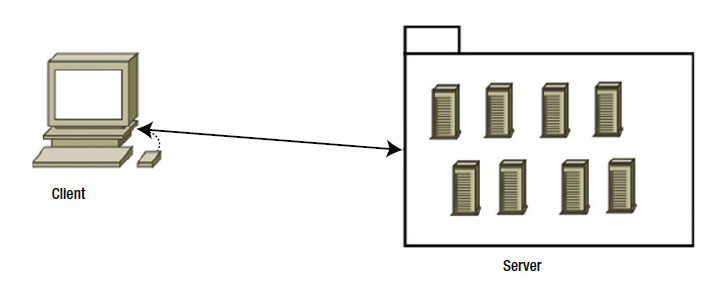
REST follows the second approach. In order to define a REST architecture, a null-state is initially defined—a system that has no constraints whatsoever and where component differentiation is nothing but a

Myth—and constraints are added one by one.

**Client-Server**

The first constraint to be added is one of the most common ones on network-based architectures: *client -*

*Server.* A server is in charge of handling a set of services, and it listens for requests regarding said services.

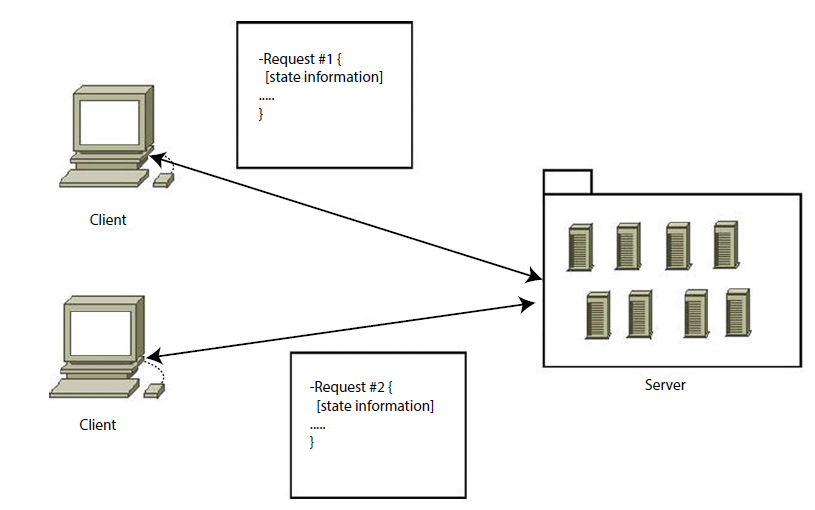


**Figure 1-1.** *Client-Server architecture diagram*

The main principle behind this constraint is the *separation of concerns.* It allows for the separation of front-end code (representation and possible UI-related processing of the information) from the server side code, which should take care of storage and server-side processing of the data. This constraint allows for the independent evolution of both components, offering a great deal of flexibility by letting client applications improve without affecting the server code and vice-versa.

**Stateless**

The constraint to be added on top of the previous one is the *stateless* constraint (see Figure 1-2 ). Communication between client and server must be stateless, meaning that each request done from the client must have all the information required for the server to understand it, without taking advantage of any stored data.



**Figure 1-2.** *Representation of the stateless constraint*

This constraint represents several improvements for the underlying architecture:

• *Visibility:* Monitoring the system becomes easy when all the information required is inside the request.

• *Scalability:* By not having to store data between requests, the server can free resources faster.

• *Reliability:* As mentioned earlier, a system that is stateless can recover from a failure much easier than one that isn’t, since the only thing to recover is the application itself.

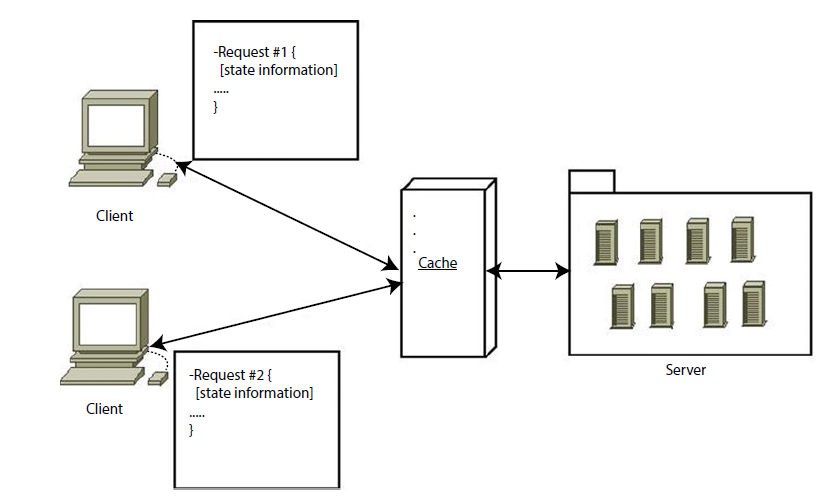
• *Easier implementation:* Writing code that doesn’t have to manage stored state data across multiple servers is much easier to do, thus the full server-side system becomes simpler.

Although at first glance this constraint might seem nothing but good, as what normally happens, there is a trade-off. On one hand, benefits are gained by the system, but on the other side, network traffic could potentially be harmed by adding a minor overhead on every request from sending repeated state information. Depending on the type of system being implemented, and the amount of repeated information, this might not be an acceptable trade-off.

**Cacheable**

The cacheable constraint is added to the current set of constraints (see Figure 1-3 ). It proposes that every

response to a request must be explicitly or implicitly set as cacheable (when applicable).



**Figure 1-3.** *Representation of a client-stateless-cache-server architecture*

By caching the responses, there are some obvious benefits that get added to the architecture: on the

server side, some interactions (a database request, for example) are completely bypassed while the content

is cached. On the client side, an apparent improvement of performance is perceived.

The trade-off with this constraint is the possibility of cached data being stale, due to poor caching rules.

This constraint is, again, dependent on the type of system being implemented.

Note Figure 1 -3 shows the cache as an external layer between the clients and the servers. This is only one possible implementation of it. The cache layer could be living inside the client (i.e., browser cache) or inside the servers themselves.

**Uniform Interface**

One of REST’s main characteristics and winning points when compared to other alternatives is the uniform

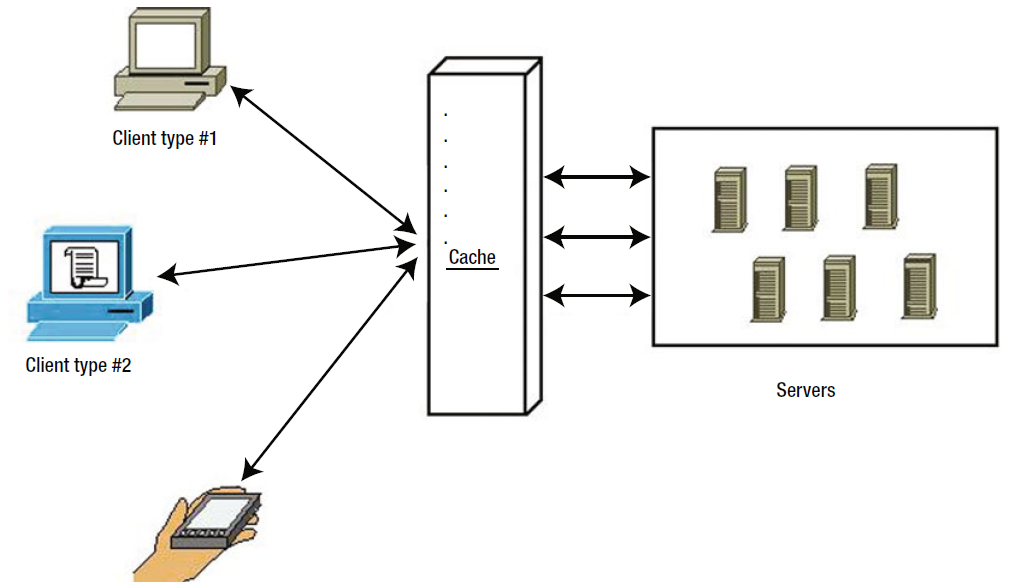
Interface constraint. By keeping a uniform interface between components, you simplify the job of the client

When it comes to interacting with your system (see Figure 1-4). Another major winning point here is that the

client’s implementation is independent of yours, so by defining a standard and uniform interface for all of

your services, you effectively simplified the implementation of independent clients by giving them a clear set

Of rules to follow.



**Figure 1-4.** *Different client types can interact seamlessly with servers thanks to the uniform interface*

**Note** In order to achieve the uniform interface, a new set of constraints must be added to the interface:

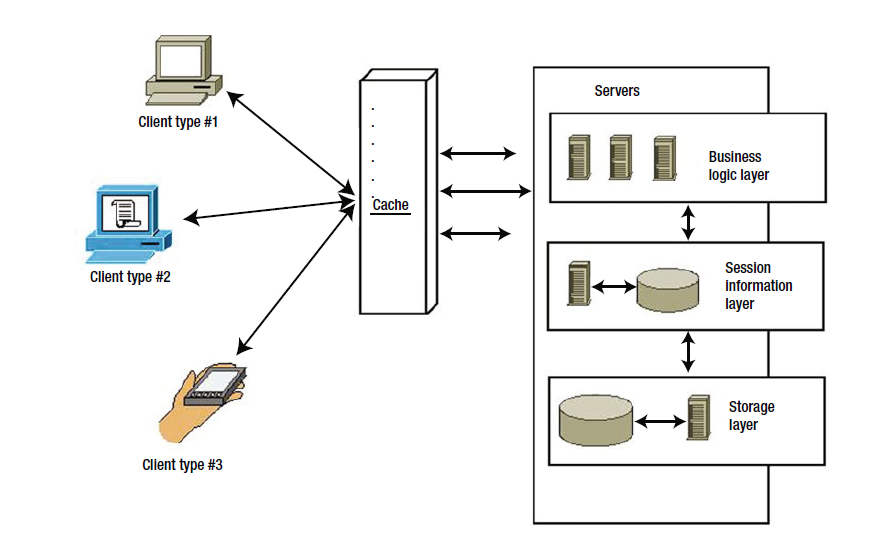
Identification of resources, manipulation of resources through representation, self-descriptive messages, and

Hypermedia as the engine of application state (a.k.a HATEOAS). I’ll discuss some of these constraints shortly.

**Layered System**

REST was designed with the Internet in mind, which means that an architecture that follows REST is

expected to work properly with the massive amount of traffic that exists in the web of webs. In order to achieve this, the concept of layers is introduced (see Figure 1-5). By separating components into layers, and allowing each layer to only use the one below and to communicate its output to the one above, you simplify the system’s overall complexity and keep component coupling in check. This is a great benefit in all type of systems, especially when the complexity of such a system is ever-growing (e.g., systems with massive amounts of clients, systems that are currently evolving, etc.).



**Figure 1-5.** *Example of a multilayered architecture*

The main disadvantage of this constraint is that for small systems, it might add unwanted latency into

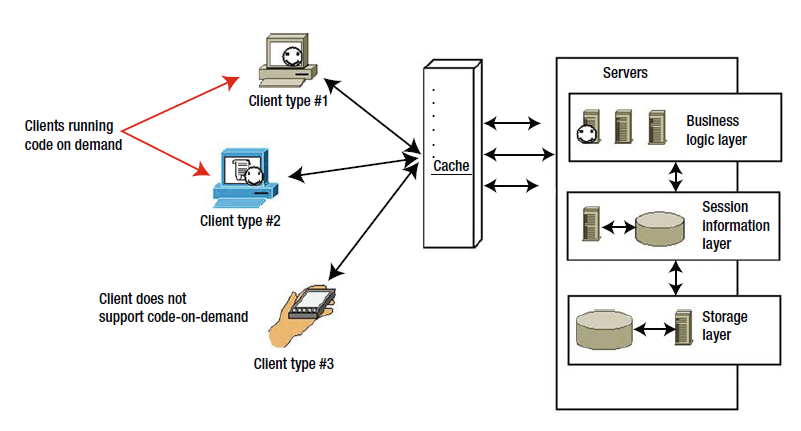
the overall data flow, due to the different interactions between layers.

**Code-on-Demand**

Code-on-demand is the only optional constraint imposed by REST, which means that an architect using REST

can choose whether or not to use this constraint, and either gains its advantages or suffers its disadvantages.

With this constraint, the client can download and execute code provided by the server (such as Java applets, JavaScript scripts, etc.). In the case of REST APIs (which is what this book focuses on), this constraint seems unnecessary, because the normal thing for an API client to do is just get information from an endpoint, and then process it however needed; but for other uses of REST, like web servers, a client (i.e., a browser) will probably benefit from this constraint (see Figure 1-6).



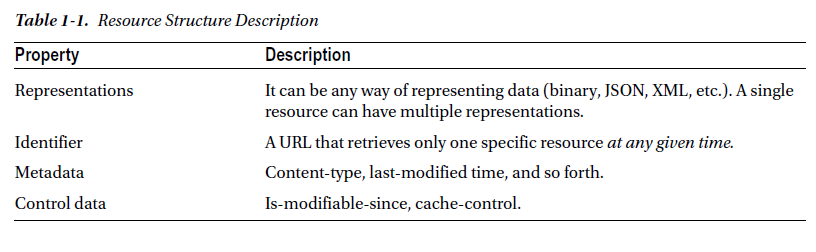
**Figure 1-6.** *How some clients might execute the code-on-demand, whereas others might not*

The main building blocks of a REST architecture are the resources. Anything that can be named can be a

Resource (a web page, an image, a person, a weather service report, etc.). Resources define what the services

Are going to be about, the type of information that is going to be transferred, and their related actions. The

Resource is the main entity from which everything else is born. A resource is the abstraction of anything that can be conceptualized (from an image file, to a plain text document). The structure of a resource is shown in Table 1-1.



**Representations**

At its core, a representation is a set of bytes, and some metadata that describes these bytes. A single resource

can have more than one representation; just think of a weather service report (which could act as a possible

resource).

The weather report for a single day could potentially return the following information:

• The date the report is referencing

• The maximum temperature for the day

• The minimum temperature for the day

• The temperature unit to be used

• A humidity percentage

• A code indicating how cloudy the day will be (e.g., high, medium, low)

Now that the resource’s structure is defined, here are a few possible representations of the same

resource:

JSON

{

"date": "2014-10-25",

"max\_temp": 25.5,

"min\_temp": 10.0,

"temp\_unit": "C",

"humidity\_percentage": 75.0,

"cloud\_coverage": "low"

}

XML

<?xml version='1.0' encoding='UTF-8' ?>

<root>

<temp\_unit value="C" />

<humidity\_percentage value="75.0" />

<cloud\_coverage value="low" />

<date value="2014-10-25" />

<min\_temp value="10.0" />

<max\_temp value="25.5" />

</root>

Custom pipe-separated values:

2014-10-25|25.5|10.0|C|75.0|low

And there could be many more

**Note**: Unless you’re doing some sort of consistency check against the API, there is no point in requesting more than one representation of the same resource, is there?

There are two very popular ways to let the client request a specific representation on a resource that has

more than one. The first one directly follows the principles described by REST (when using HTTP as a basis),

called content negotiation, which is part of the HTTP standard. The second one is a simplified version of this,

with limited benefits. For the sake of completeness, I’ll quickly go over them both.

**Content Negotiation**

As mentioned, this methodology is part of the HTTP standard, so it’s the preferred way according to REST

(at least when focused on API development on top of HTTP). It is also more flexible and provides further

advantages than the other method.

It consists of the client sending a specific header with the information of the different content types

(or types of representations) supported, with an optional indicator of how much supported/preferred that

format is. Let’s look at an example from the “Content Negotiation” page on Wikipedia:

Accept: text/html; q=1.0, text/\*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/\*;

q=0.5, \*/\*; q=0.1

This example is from a browser configured to accept various types of resources, but preferring HTML over

plain text, and GIF or JPEG images over other types, but ultimately accepts any other content type as a last resort. On the server side, the API is in charge of reading this header and finding the best representation for

each resource, based on the client’s preferences.

**Using File Extensions**

Even though this approach is not part of the REST proposed style, it is widely used and a fairly simple

alternative to the somewhat more complex other option, so I’ll cover it anyway. During the last few years, using file extensions has become an alternative preferred over using content negotiation; it is a simpler version and it doesn’t rely on a header being sent, but instead, it works with the concept of file extensions. The extension portion of the file’s name indicates the content type to the operating system and any other software trying to use it; so in the following case, the extension added to the resource’s URL (unique identifier) indicates to the server the type of representation wanted.

GET /api/v1/books .json

GET /api/v1/books .xml

Both identifiers reference the same resource—the list of books, but they request a different representation of it.

**Note** This approach might seem easier to implement, and even understand, by humans, but it lacks the

flexibility added by content negotiation, and should only be used if there is no real need for complex cases

where multiple content types might be specified with their related preference.

**Resource Identifier**

The resource identifier should provide a unique way of identification at any given moment and it should

provide the full path to the resource. A classic mistake is to assume it’s the resource’s ID on the storage

medium used (i.e., the ID on the database). This means that you cannot consider a simple numeric ID as

a resource identifier; you must provide the full path, and because we’re basing REST on HTTP, the way to

access the resource it to provide its full URI (unique resource identifier).

There is one more aspect to consider: the identifier of each resource must be able to reference it

unequivocally at any given moment in time. This is an important distinction, because a URI like the

following might reference Harry Potter and the Half Blood Prince for a certain period of time, and then

Harry Potter and the Deathly Hollows one year later.

GET /api/v1/books/last

This renders that URI as an invalid resource ID. Instead, each book needs a unique URI that is certain to

not change over time; for example:

GET /api/v1/books /j-k-rowling/harry-potter-and-the-deathly-hollows

GET /api/v1/books /j-k-rowling/harry-potter-and-the-half-blood-prince

The identifiers are unique here, because you can safely assume that the author won’t publish more

books with the same title.And to provide a valid example for getting the last book, you might consider doing something like this:

GET /api/v1/books?limit=1&sort=created\_at

The preceding URI references the lists of books, and it asks for only one, sorted by it’s publish date, thus

rendering the last book added.

**Actions**

Identifying a resource is easy: you know how to access it and you even know how to request for a specific

format (if there is more than one); but that’s not all that REST proposes. Since REST is using the HTTP

protocol as a standing point, the latter provides a set of verbs that can be used to reference the type of action

being done over a resource.

There are other actions, aside from accessing, that a client app can take in the resources provided by an

API; these depend on the service provided by the API. These actions could potentially be anything, just like

the type of resources handled by the system. Still, there is a set of common actions that any system that is

resource-oriented should be able to provide: CRUD (create, retrieve, update, and delete) actions.

These so-called actions can be directly mapped to the HTTP verbs, but REST does not enforce a

standardized way to do so. However, there are some actions that are naturally derived by the verb and others

that have been standardized by the API development community over the years, as shown in Table 1-2.

