Representational State Transfer Architectural Style

Internet of Things

Rahul Shandilya

REST: Introduction

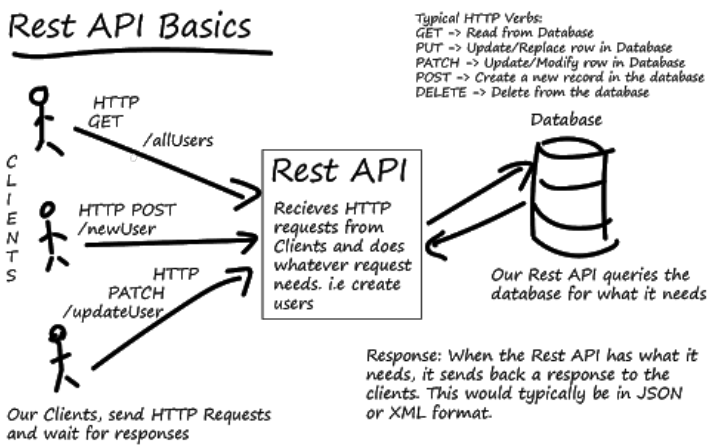
REpresentational State Transfer (REST) is the architectural style behind the web. Defined in 2000 in Roy Fielding's PhD thesis, REST defines a set of rules and principles that all the elements of the architecture must conform to in order to build web applications that scale well, in terms of:

- scalability (number of interacting clients)
- robustness (long-term evolution of systems).

REST: Introduction

REpresentational State Transfer (REST) is the architectural style behind the web. Defined in 2000 in Roy Fielding's PhD thesis, REST defines a set of rules and principles that all the elements of the architecture must conform to in order to build web applications that scale well, in terms of:

- scalability (number of interacting clients)
- robustness (long-term evolution of systems).



REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements.

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions
- *servers* (origin servers) host resources and serve client requests.

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions
- *servers* (origin servers) host resources and serve client requests.

A REST architecture is characterized by uniform interfaces: all connectors within the system must conform to this interface's constraints. Collectively, REST defines the following principles:

- identification of resources

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions
- *servers* (origin servers) host resources and serve client requests.

A REST architecture is characterized by uniform interfaces: all connectors within the system must conform to this interface's constraints. Collectively, REST defines the following principles:

- identification of resources
- manipulation of resources through representations

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions
- *servers* (origin servers) host resources and serve client requests.

A REST architecture is characterized by uniform interfaces: all connectors within the system must conform to this interface's constraints. Collectively, REST defines the following principles:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages

REST Architectures

The principle of separation of concerns is a fundamental of the REST architecture. According to this principle, each element of a system is responsible for a specific concern. Well-separated concerns allow for modularity, reusability, and independent evolution of the system's elements. A REST architecture builds on:

- *clients* (or user agents, such as browsers), which are the application interface and initiate the interactions
- *servers* (origin servers) host resources and serve client requests.

A REST architecture is characterized by uniform interfaces: all connectors within the system must conform to this interface's constraints. Collectively, REST defines the following principles:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application states.

Representation of Resources

Resources are never exchanged directly by endpoints. Instead, representations of resources are exchanged between endpoints. A representation is a view of the state of the resource at a given time.

Representation of Resources

Resources are never exchanged directly by endpoints. Instead, representations of resources are exchanged between endpoints. A representation is a view of the state of the resource at a given time.

- This view can be encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.

Representation of Resources

Resources are never exchanged directly by endpoints. Instead, representations of resources are exchanged between endpoints. A representation is a view of the state of the resource at a given time.

- This view can be encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.
- Typically, the type of representation is specified in one header of the message containing the resource; for example, HTTP defines the Content-Type header.

Representation of Resources

Resources are never exchanged directly by endpoints. Instead, representations of resources are exchanged between endpoints. A representation is a view of the state of the resource at a given time.

- This view can be encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.
- Typically, the type of representation is specified in one header of the message containing the resource; for example, HTTP defines the Content-Type header.
- Resources are exchanged back and forth between clients and servers. In order to be exchanged, resources must be serialized/deserialized properly at each endpoint.

Representation of Resources

Resources are never exchanged directly by endpoints. Instead, representations of resources are exchanged between endpoints. A representation is a view of the state of the resource at a given time.

- This view can be encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.
- Typically, the type of representation is specified in one header of the message containing the resource; for example, HTTP defines the Content-Type header.
- Resources are exchanged back and forth between clients and servers. In order to be exchanged, resources must be serialized/deserialized properly at each endpoint.
- The same resource can have many different representations (1:N relationship): the state of the same sensor can be described using JSON, XML, or any other suitable format.

Resource Identifiers

In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

Resource Identifiers

In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

- A URI identifies a resource univocally. A URI can be used to address a resource, so that it can be located, retrieved, and manipulated.

Resource Identifiers

In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

- A URI identifies a resource univocally. A URI can be used to address a resource, so that it can be located, retrieved, and manipulated.
- There is a 1:N relationship between a resource and URIs: a resource can be mapped to multiple URIs, but a URI points exactly to one resource.

Resource Identifiers

In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

- A URI identifies a resource univocally. A URI can be used to address a resource, so that it can be located, retrieved, and manipulated.
- There is a 1:N relationship between a resource and URIs: a resource can be mapped to multiple URIs, but a URI points exactly to one resource.
- URIs can be of two kinds: (i). a uniform resource name (URN) specifies the name of a resource (e.g.,urn:ietf:rfc:2616);

Resource Identifiers

In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

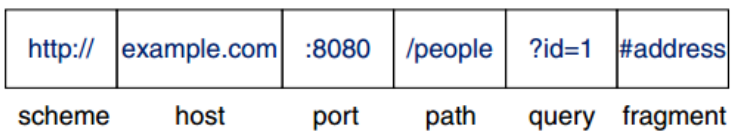
- A URI identifies a resource univocally. A URI can be used to address a resource, so that it can be located, retrieved, and manipulated.
- There is a 1:N relationship between a resource and URIs: a resource can be mapped to multiple URIs, but a URI points exactly to one resource.
- URIs can be of two kinds: (i). a uniform resource name (URN) specifies the name of a resource (e.g.,urn:ietf:rfc:2616);
- a uniform resource locator (URL) specifies how to locate the resource, (e.g., http://example.com/books/123).

Resource Identifiers

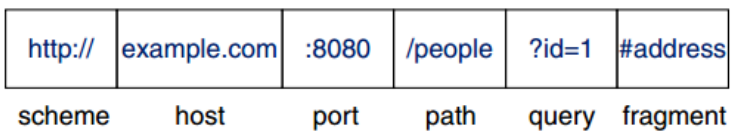
In order to ensure that an application is handling the correct resource, a mechanism to identify a resource univocally in the network is necessary. uniform resource identifiers (URIs), defined in RFC 3986, serve this specific need.

- A URI identifies a resource univocally. A URI can be used to address a resource, so that it can be located, retrieved, and manipulated.
- There is a 1:N relationship between a resource and URIs: a resource can be mapped to multiple URIs, but a URI points exactly to one resource.
- URIs can be of two kinds: (i). a uniform resource name (URN) specifies the name of a resource (e.g.,urn:ietf:rfc:2616);
- a uniform resource locator (URL) specifies how to locate the resource, (e.g., http://example.com/books/123).
- All URIs take the following form: scheme:scheme-specificpart. The scheme part defines how the rest of the URI is to be interpreted – it typically serves as an indication of the communication protocol that should be used to target the resource.

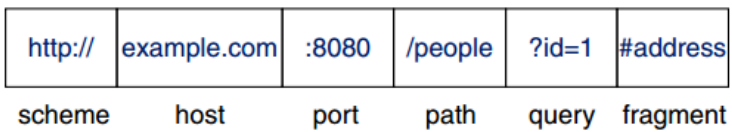
- URLs include all the information needed to successfully address the resource. The host and optional port include networking information needed to reach to the resource.



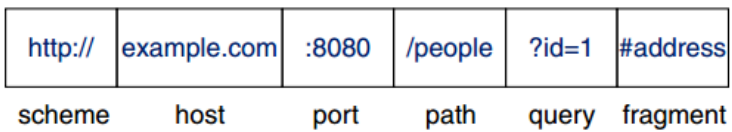
- URLs include all the information needed to successfully address the resource. The host and optional port include networking information needed to reach to the resource.
- The host can be be either an IP address or a fully qualified domain name, which must be resolved using the DNS system.



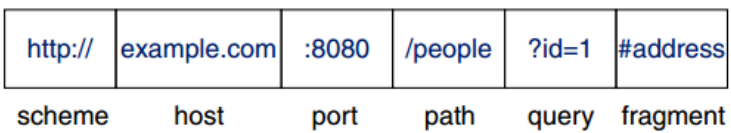
- URLs include all the information needed to successfully address the resource. The host and optional port include networking information needed to reach to the resource.
- The host can be be either an IP address or a fully qualified domain name, which must be resolved using the DNS system.
- The path provides information to locate the resource inside the host. The query contains matching information to filter out the result.



- URLs include all the information needed to successfully address the resource. The host and optional port include networking information needed to reach to the resource.
- The host can be be either an IP address or a fully qualified domain name, which must be resolved using the DNS system.
- The path provides information to locate the resource inside the host. The query contains matching information to filter out the result.
- The fragment can be used to identify a specific portion of the resource.



- URLs include all the information needed to successfully address the resource. The host and optional port include networking information needed to reach to the resource.
- The host can be be either an IP address or a fully qualified domain name, which must be resolved using the DNS system.
- The path provides information to locate the resource inside the host. The query contains matching information to filter out the result.
- The fragment can be used to identify a specific portion of the resource.
- URIs should be opaque and not expose any specific notion of the format used to represent the targeted resource.



Statelessness

An important principle of REST is statelessness. Statelessness implies that no state information must be kept on the client and server sides, thus avoiding the need to use cookies or to introduce the concept of sessions, which demand a stricter coupling between the endpoints.

Statelessness

An important principle of REST is statelessness. Statelessness implies that no state information must be kept on the client and server sides, thus avoiding the need to use cookies or to introduce the concept of sessions, which demand a stricter coupling between the endpoints.

- All requests must therefore be stateless.

Statelessness

An important principle of REST is statelessness. Statelessness implies that no state information must be kept on the client and server sides, thus avoiding the need to use cookies or to introduce the concept of sessions, which demand a stricter coupling between the endpoints.

- All requests must therefore be stateless.
- In order to preserve statelessness, each message must be self-descriptive. This means that all requests must contain all the information to understand the request so that servers can process them without context (about the state of the client).

Statelessness

An important principle of REST is statelessness. Statelessness implies that no state information must be kept on the client and server sides, thus avoiding the need to use cookies or to introduce the concept of sessions, which demand a stricter coupling between the endpoints.

- All requests must therefore be stateless.
- In order to preserve statelessness, each message must be self-descriptive. This means that all requests must contain all the information to understand the request so that servers can process them without context (about the state of the client).
- There is no state information maintained between clients and servers: the state of the client is contained in the request, so the server is relieved from the burden of keeping track of the client state.

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

- Due to the loose coupling between clients and servers, the states of the FSM and the transitions between them are not known in advance.

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

- Due to the loose coupling between clients and servers, the states of the FSM and the transitions between them are not known in advance.
- A state is reached when the server transfers a representation of the resource as a consequence of a client request.

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

- Due to the loose coupling between clients and servers, the states of the FSM and the transitions between them are not known in advance.
- A state is reached when the server transfers a representation of the resource as a consequence of a client request.
- The next possible transitions are discovered when the application reaches a new state (gradual reveal).

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

- Due to the loose coupling between clients and servers, the states of the FSM and the transitions between them are not known in advance.
- A state is reached when the server transfers a representation of the resource as a consequence of a client request.
- The next possible transitions are discovered when the application reaches a new state (gradual reveal).
- Resource representations that embed links are called hypermedia. These links represent the possible transitions to the next states.

Applications as Finite-state Machines

RESTful applications make forward progress by transitioning from one state to another, just like a finite-state machine (FSM)

- Due to the loose coupling between clients and servers, the states of the FSM and the transitions between them are not known in advance.
- A state is reached when the server transfers a representation of the resource as a consequence of a client request.
- The next possible transitions are discovered when the application reaches a new state (gradual reveal).
- Resource representations that embed links are called hypermedia. These links represent the possible transitions to the next states.
- In essence, the state of the a resource identified by a URI is contained in the data section of the resource representation and the transition to the next states are contained in the links.

Hypermedia as the Engine of Application State

The final principle of REST is “hypermedia as the engine of application state”, or HATEOAS for short.

Hypermedia as the Engine of Application State

The final principle of REST is "hypermedia as the engine of application state", or HATEOAS for short.

- HATEOAS states that resource representation should include all the information to drive the client to perform the next requests to progress in the application.

Hypermedia as the Engine of Application State

The final principle of REST is "hypermedia as the engine of application state", or HATEOAS for short.

- HATEOAS states that resource representation should include all the information to drive the client to perform the next requests to progress in the application.
- A client just needs to follow the instructions that the server transmits in order to reach its goal.

Hypermedia as the Engine of Application State

The final principle of REST is "hypermedia as the engine of application state", or HATEOAS for short.

- HATEOAS states that resource representation should include all the information to drive the client to perform the next requests to progress in the application.
- A client just needs to follow the instructions that the server transmits in order to reach its goal.
- This guarantees that, should the server change its implementation and introduce new functionality (states and links in the FSM), the client would be unaffected by these changes and could continue to operate.

Hypertext Transport Protocol (HTTP) Verbs

Action	Verb	Result
Create Data	POST	create a *resource* when you don't know URI, partial update of a *resource*, invoke arbitrary processing
Read Data	GET	retrieve a *representation* of the *resource*
Update Data	PUT	create a *resource* at a known URI or update an existing *resource* by replacing it
Delete Data	DELETE	delete a *resource*
List Data	GET	*perform get on a container using a path hierarchy*

In summary, RESTful applications progress according to the following steps:

- The client starts from an entry URI or a bookmark.

In summary, RESTful applications progress according to the following steps:

- The client starts from an entry URI or a bookmark.
- The response to a GET request includes a hypermedia representation.

In summary, RESTful applications progress according to the following steps:

- The client starts from an entry URI or a bookmark.
- The response to a GET request includes a hypermedia representation.
- The representation contains links that define the possible transitions to the next states of the FSM.

In summary, RESTful applications progress according to the following steps:

- The client starts from an entry URI or a bookmark.
- The response to a GET request includes a hypermedia representation.
- The representation contains links that define the possible transitions to the next states of the FSM.
- The client selects a link to follow and issues the next request; that is, it triggers a transition to the next state.

In summary, RESTful applications progress according to the following steps:

- The client starts from an entry URI or a bookmark.
- The response to a GET request includes a hypermedia representation.
- The representation contains links that define the possible transitions to the next states of the FSM.
- The client selects a link to follow and issues the next request; that is, it triggers a transition to the next state.
- The client can also go back.

Advantage of REST

RESTful Web services are easier to learn, more intuitive and more suitable for programming IoT applications. The main advantages of REST are:

- Simplicity - Since REST uses standard HTTP it is much simpler demanding minimal tooling/middleware (only HTTP support is required);

Advantage of REST

RESTful Web services are easier to learn, more intuitive and more suitable for programming IoT applications. The main advantages of REST are:

- Simplicity - Since REST uses standard HTTP it is much simpler demanding minimal tooling/middleware (only HTTP support is required);
- The variety of data formats - REST permits many different data formats while SOAP permits XML only. In such a way REST allows better support for browser clients due to JSON (JavaScript Object Notation) support, which usually a better fits data and parses much faster;

Advantage of REST

RESTful Web services are easier to learn, more intuitive and more suitable for programming IoT applications. The main advantages of REST are:

- Simplicity - Since REST uses standard HTTP it is much simpler demanding minimal tooling/middleware (only HTTP support is required);
- The variety of data formats - REST permits many different data formats while SOAP permits XML only. In such a way REST allows better support for browser clients due to JSON (JavaScript Object Notation) support, which usually a better fits data and parses much faster;
- Performance and scalability - REST has better performance and scalability characteristics.

Advantage of REST

RESTful Web services are easier to learn, more intuitive and more suitable for programming IoT applications. The main advantages of REST are:

- Simplicity - Since REST uses standard HTTP it is much simpler demanding minimal tooling/middleware (only HTTP support is required);
- The variety of data formats - REST permits many different data formats while SOAP permits XML only. In such a way REST allows better support for browser clients due to JSON (JavaScript Object Notation) support, which usually a better fits data and parses much faster;
- Performance and scalability - REST has better performance and scalability characteristics.
- REST architecture has intuitiveness, flexibility, better cache support, lightweight requests and responses, and easier response parsing.

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via “RESTful” interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via "RESTful" interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

- Resource - anything that's important enough to be referenced as a thing itself;

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via "RESTful" interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

- Resource - anything that's important enough to be referenced as a thing itself;
- Resource name - unique identification of the resource;

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via "RESTful" interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

- Resource - anything that's important enough to be referenced as a thing itself;
- Resource name - unique identification of the resource;
- Resource representation - useful information about the current state of a resource;

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via "RESTful" interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

- Resource - anything that's important enough to be referenced as a thing itself;
- Resource name - unique identification of the resource;
- Resource representation - useful information about the current state of a resource;
- Resource link - link to another representation of the same or other resource;

REST as ROA

A Resource-Oriented Architecture (ROA) presents an abstraction of resources implemented via "RESTful" interfaces . RESTful platforms, based on REST development technologies, enable the creation of ROA whose main concepts are:

- Resource - anything that's important enough to be referenced as a thing itself;
- Resource name - unique identification of the resource;
- Resource representation - useful information about the current state of a resource;
- Resource link - link to another representation of the same or other resource;
- Resource interface - uniform interface for accessing the resource and manipulating its state.

Smart Things as Resources

IoT can be defined as a worldwide network of interconnected “smart things” enabled to interact and communicate to each other, as well as with the environment, by exchanging data and information “sensed” from the environment.

Smart Things as Resources

IoT can be defined as a worldwide network of interconnected "smart things" enabled to interact and communicate to each other, as well as with the environment, by exchanging data and information "sensed" from the environment. Three main system-level characteristics of IoT [2] are:

- Anything communicates: smart things have the ability to wirelessly communicate among themselves, and form ad-hoc networks of interconnected objects;

Smart Things as Resources

IoT can be defined as a worldwide network of interconnected "smart things" enabled to interact and communicate to each other, as well as with the environment, by exchanging data and information "sensed" from the environment. Three main system-level characteristics of IoT [2] are:

- Anything communicates: smart things have the ability to wirelessly communicate among themselves, and form ad-hoc networks of interconnected objects;
- Anything is identified: smart things are identified with a digital name; and

Smart Things as Resources

IoT can be defined as a worldwide network of interconnected "smart things" enabled to interact and communicate to each other, as well as with the environment, by exchanging data and information "sensed" from the environment. Three main system-level characteristics of IoT [2] are:

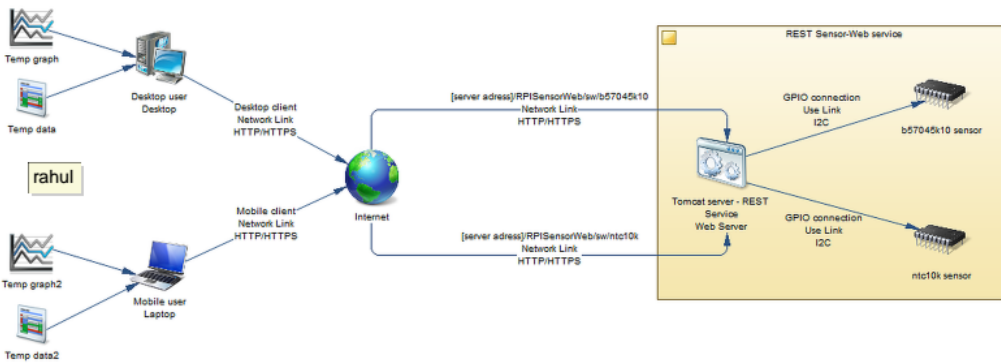
- Anything communicates: smart things have the ability to wirelessly communicate among themselves, and form ad-hoc networks of interconnected objects;
- Anything is identified: smart things are identified with a digital name; and
- Anything interacts: smart things can interact with the local environment through sensing and actuation capabilities whenever present.

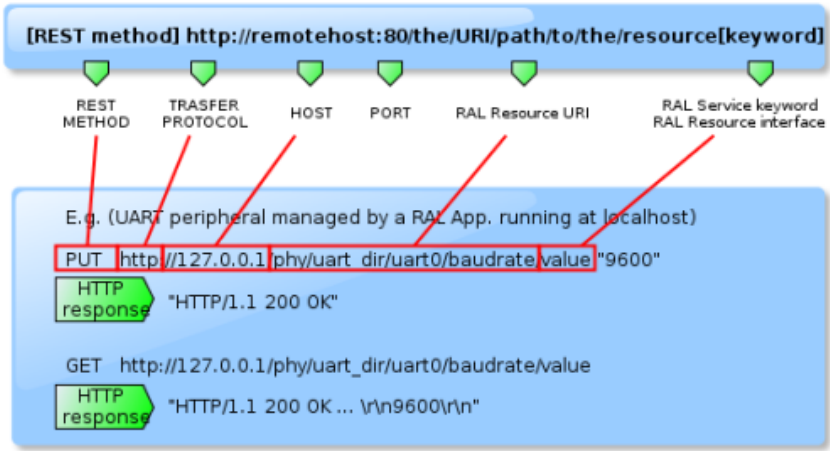
Smart things can be made as an integral part of the Web by their embedding into a standardized Web service architecture. Thus, the term "Web of Things", or WoT, can be considered as a part or a subset of IoT which incorporates similar characteristics and application models as IoT.

Redefining Resource for IOT

As specified by RFC 3986, any entity that has an identity can be a resource, even abstract concepts such as mathematical operators in an equation.
We can redefine resource to incorporate Smart Things.
"A Web resource is any real or abstract entity that can be identified and accessed within the IP network. A Web resource gives access to a specific functionality, or is part of a composition of other resources to provide a well-defined network application functionality."

IOT as REST architecture




[REST method] http://remotehost:80/the/URI/path/to/the/resource[keyword]
REST METHOD
TRASFER PROTOCOL
HOST
PORT
RAL Resource URI
RAL Service keyword
RAL Resource interface
E.g. (UART peripheral managed by a RAL App. running at localhost)
PUT http://127.0.0.1/phy/uart_dir/uart0/baudrate/value "9600"
HTTP response
"HTTP/1.1 200 OK"
GET http://127.0.0.1/phy/uart_dir/uart0/baudrate/value
HTTP response
"HTTP/1.1 200 OK ... \r\n9600\r\n"