rest

💻 Summary of REST (Representational State Transfer Protocol)

Recall

- What is REST?
- What are the six guiding principles?

Overview

REST is an acronym for **RE**presentational **S**tate **T**ransfer and an architectural style for **distributed hypermedia systems.** REST has its guiding principles and constraints.

These principles must be satisfied if a service interface needs to be referred to as **RESTful.**

📘 A Web API (or Web Service) conforming to the REST architectural style is a *REST API.*

Guiding Principles

Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

The following four constraints can achieve a uniform REST interface:

- **Identification of resources** – The interface must uniquely identify each resource involved in the interaction between the client and the server.
- **Manipulation of resources through representations** – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
- **Self-descriptive messages** – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
- **Hypermedia as the engine of application state** – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

Client-Server

The client-server design pattern enforces the **separation of concerns**, which helps the client and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

Layered System

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

Code on Demand (optional)

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

Resources

The key **abstraction of information** in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person).

The state of the resource, at any particular time, is known as the **resource representation**.

The resource representations are consist of:

- the **data**
- the **metadata** describing the data
- and the **hypermedia links** that can help the clients in transition to the next desired state.

> 📘 A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API's ***resource model***.

Resource Identifiers

REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

Hypermedia

The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed.

A RESTful API looks like *hypertext*. Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

> *Hypertext* (or hypermedia) means the **simultaneous presentation of information and controls** such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.

> Remember that hypertext does not need to be HTML (or XML or JSON) on a browser. Machines can follow links when they understand the data format and relationship types.
>
> — *Roy Fielding*

Self-Descriptive

Further, **resource representations shall be self-descriptive**: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource.

So in practice, we will create lots of **custom media types** – usually one media type associated with one resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

> 📘 Media Types have no relation to the resource methods GET/PUT/POST/DELETE/… other than the fact that some media type elements will define a process model that goes like “anchor elements with an `href` attribute create a hypertext link that, when selected, invokes a retrieval request (GET) on the URI corresponding to the `CDATA`-encoded `href` attribute.”

Resource Methods

Another important thing associated with REST is **resource methods**. These resource methods are used to perform the desired transition between two states of any resource.

A large number of people wrongly relate resource methods to HTTP methods (i.e., GET/PUT/POST/DELETE). Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be a **uniform interface**.

For example, if we decide that the application APIs will use HTTP POST for updating a resource – rather than most people recommend HTTP PUT – it’s all right. Still, the application interface will be RESTful.

Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

> We should enter a REST API with no prior knowledge beyond the initial URI (a bookmark) and a set of standardized media types appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).
>
> From that point on, all application state transitions must be driven by the client selection of server-provided choices present in the received representations or implied by the user’s manipulation of those representations.
>
> The transitions may be determined (or limited by) the client’s knowledge of media types and resource communication mechanisms, both of which may be improved on the fly (e.g., *code-on-demand*). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

REST ≠ HTTP

> 📘 Many people prefer to compare HTTP with REST. **REST and HTTP are not the same.**

Though REST also intends to make the web (internet) more streamlined and standard, Roy Fielding advocates using REST principles more strictly. And that’s from where people try to start comparing REST with the web.

Roy Fielding, in his dissertation, has nowhere mentioned any implementation direction – including any protocol preference or even HTTP. Till the time, we are honoring the six guiding principles of REST, which we can call our interface – RESTful.

Summary

In simple words, in the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers** (URIs).

The resources are acted upon by using a set of simple, well-defined operations. Also, the resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

The clients and servers exchange representations of resources by using a standardized interface and protocol. Typically HTTP is the most used protocol, but REST does not mandate it.

Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

And most importantly, every interaction with the server must be stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.