表徵狀態轉移

表述性狀態轉移(REST) 是一種軟件架構風格，它描述了物理上獨立的組件之間的統一接口，通常在客戶端-服務器架構中跨越Internet 。REST 定義了四種接口約束：

- 資源識別
- 操縱資源
- 自描述消息和
- 超媒體作為應用程序狀態的引擎[1]

通常 REST 描述了機器對機器的接口。在 Web 開發中，REST 允許在請求時呈現內

容，通常稱為動態內容。RESTful 動態內容使用服務器端呈現來生成網站並將內容髮送到發出請求的 Web 瀏覽器，瀏覽器解釋服務器的代碼並在用戶的 Web 瀏覽器中呈現頁面。

REST 已在整個軟件行業得到廣泛應用，並被廣泛接受為一套用於創建無狀態、可靠的Web API的指南。遵守 REST 約束的 Web API 被非正式地描述為*RESTful*。通常，RESTful Web API 鬆散地基於HTTP方法，例如 GET 和 POST。HTTP 請求用於通過URL 編碼的參數訪問 Web 應用程序中的數據或資源。響應通常採用JSON或XML 格式來傳輸數據。

“Web 資源”最初在萬維網上被定義為由其URL標識的文檔或文件。如今，該定義更加通用和抽象，包括連接到 Internet、本地網絡或設備的所有事物、實體或操作。Internet 上的每個設備都有一個 URI 或統一資源標識符。在 RESTful Web 服務中，對資源的URI發出的請求會引發一個響應，其中的負載格式為HTML、XML、JSON或其他格式。這些請求和響應最常用的協議是 HTTP，它提供了諸如 OPTIONS、GET、POST、PUT、PATCH 和 DELETE 等操作（ HTTP 方法）。[2][3] 通過使用無狀態協議和標準操作，RESTful 系統旨在通過重用可以管理和更新的組件來實現快速性能、可靠性和增長能力，而不會影響整個系統，即使它是跑步。

詞源

The term *representational state transfer* was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.[1][4] The term is intended to evoke an image of how a well-designed Web application behaves: it is a network of Web resources (a virtual state machine) where the user advances through the application by selecting links (e.g. http://www.example.com/articles/21), resulting in the next resource's representation (the next application state) being transferred to the client and rendered for the user.

歷史

Roy Fielding speaking at OSCON 2008

The Web began to enter everyday use in 1993–1994, when websites for general use started to become available.[5] At the time, there was only a fragmented description of the Web's architecture, and there was pressure in the industry to agree on some standard for the Web

interface protocols. For instance, several experimental extensions had been added to the communication protocol (HTTP) to support proxies, and more extensions were being proposed, but there was a need for a formal Web architecture with which to evaluate the impact of these changes.[6]

The W3C and IETF working groups together started work on creating formal descriptions of the Web's three primary standards: URI, HTTP, and HTML. Roy Fielding was involved in the creation of these standards (specifically HTTP 1.0 and 1.1, and URI), and during the next six years he developed the REST

architectural style, testing its constraints on the Web's protocol standards and using it as a means to define architectural improvements — and to identify architectural mismatches. Fielding defined REST in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" at UC Irvine.

To create the REST architectural style, Fielding identified the requirements that apply when creating a world-wide network-based application, such as the need for a low entry barrier to enable global adoption. He also surveyed many existing architectural styles for network-

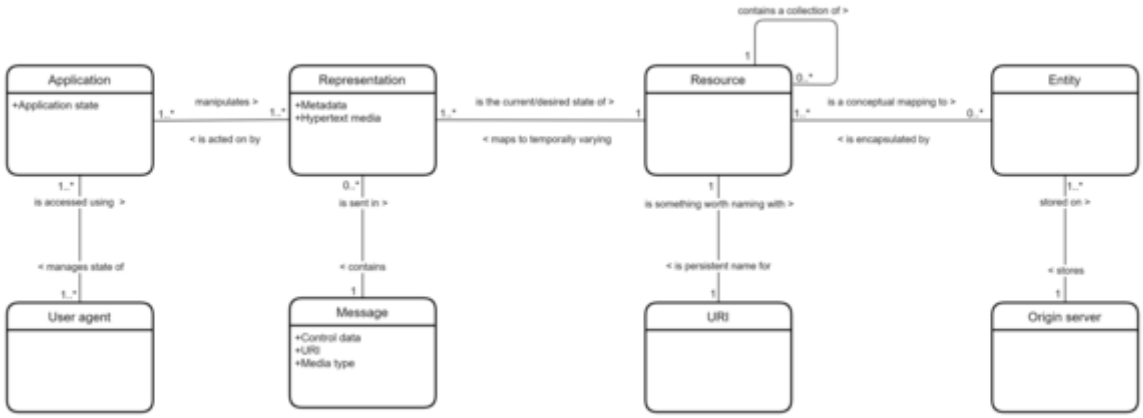
based applications, identifying which features are shared with other styles, such as caching and client–server features, and those which are unique to REST, such as the concept of resources. Fielding was trying to both categorise the existing architecture of the current implementation and identify which aspects should be considered central to the behavioural and performance requirements of the Web.

By their nature, architectural styles are independent of any specific implementation, and while REST was created as part of the development of the Web standards, the implementation of

the Web does not obey every constraint in the REST architectural style. Mismatches can occur due to ignorance or oversight, but the existence of the REST architectural style means that they can be identified before they become standardised. For example, Fielding identified the embedding of session information in URIs as a violation of the constraints of REST which can negatively affect shared caching and server scalability. HTTP cookies also violated REST constraints because they can become out of sync with the browser's application state, making them unreliable; they also contain opaque data

that can be a concern for privacy and security.

建築理念



The REST architectural style is designed for network-based applications, specifically client-server applications. But more than that, it is designed for Internet-scale usage, so the coupling between the **user agent** (client) and the **origin server** must be as lightweight (loose) as possible to facilitate large-scale adoption. This is

achieved by creating a layer of abstraction on the server by defining **resources** that encapsulate **entities** (e.g. files) on the server and so hiding the underlying implementation details (file server, database, etc.). But the definition is even more general than that: any information that can be named can be a resource: an image, a database query, a temporal service (e.g. “today’s weather in London”), or even a collection of other resources. This approach allows the greatest interoperability between clients and servers in a long-lived Internet-scale environment which crosses organisational (trust) boundaries.

Clients can only access resources using **URIs**. In other words, the client requests a resource using a URI and the server responds with a **representation** of the resource. A representation of a resource is another important concept in REST; to ensure responses can be interpreted by the widest possible number of client **applications** a representation of the resource is sent in hypertext format. Thus, a resource is manipulated through hypertext representations transferred in **messages** between the clients and servers.

The strong decoupling of client and server together with the text-based

transfer of information using a uniform addressing protocol provided the basis for meeting the requirements of the Web: robustness (anarchic scalability), independent deployment of components, large-grain data transfer, and a low-entry barrier for content readers, content authors and developers alike.

建築屬性

The constraints of the REST architectural style affect the following architectural properties:[1][7]

- performance in component interactions, which can be the

dominant factor in user-perceived performance and network efficiency;[8]

- scalability allowing the support of large numbers of components and interactions among components;
- simplicity of a uniform interface;
- modifiability of components to meet changing needs (even while the application is running);
- visibility of communication between components by service agents;
- portability of components by moving program code with the data;
- reliability in the resistance to failure at the system level in the presence of

failures within components, connectors, or data.[8]

架構限制

The REST architectural style defines six guiding constraints.[7][9] When these constraints are applied to the system architecture, it gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[1] A system that complies with some or all of these constraints is loosely referred to as RESTful.

The formal REST constraints are as follows:

Client–server architecture

The client-server design pattern enforces the principle of separation of concerns: separating the user interface concerns from the data storage concerns. Portability of the user interface is thus improved. In the case of the Web, a plethora of web browsers have been developed for most platforms without the need for knowledge of any server implementations. Separation also simplifies the server components, improving scalability, but more importantly it allows components to evolve independently (anarchic scalability), which is necessary in an

Internet-scale environment that involves multiple organisational domains.

Statelessness

In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, increasing

performance by removing server load caused by retention of session information.

Cacheability

As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance. The cache can be

performed at the client machine in memory or browser cache storage. Additionally cache can be stored in a Content Delivery Network (CDN).

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client and server, it won't affect their communications, and there won't be a need to update the client or server code. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. Also,

security can be added as a layer on top of the web services, separating business logic from security logic.[10] Adding security as a separate layer enforces security policies. Finally, intermediary servers can call multiple other servers to generate a response to the client.

Code on demand (optional)

Servers can temporarily extend or customize the functionality of a client by transferring executable code: for example, compiled components such as Java applets, or client-side scripts such as JavaScript.

Uniform interface

The uniform interface constraint is fundamental to the design of any RESTful system.[1] It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

- Resource identification in requests - Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the

server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.

- Resource manipulation through representations - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state.
- Self-descriptive messages - Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.[1]

- Hypermedia as the engine of application state (HATEOAS) - Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.[11]

分類模型

Several models have been developed to help classify REST APIs according to their adherence to various principles of REST design, such as the Richardson Maturity Model.[12]

應用於網絡服務

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs.[13] HTTP-based RESTful APIs are defined with the following aspects:[14]

- a base URI, such as `http://api.example.com/`;

- standard HTTP methods (e.g., GET, POST, PUT, and DELETE);
- a media type that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json,[14] :91–99 etc.). The current representation tells the client how to compose requests for transitions to all the next available application states. This could be as simple as a URI or as complex as a Java applet.[15]

Semantics of HTTP methods

The following table shows how HTTP methods are intended to be used in HTTP APIs, including RESTful ones.

Semantics of HTTP methods

HTTP method	Description
GET[16]:§4.3.1	Get a representation of the target resource's state.
POST[16]:§4.3.3	Let the target resource process the representation enclosed in the request.
PUT[16]:§4.3.4	Create or replace the state of the target resource with the state defined by the representation enclosed in the request.
PATCH[3]	Partially update resource's state.
DELETE[16]:§4.3.5	Delete the target resource's state.
OPTIONS[16]:§4.3.7	Advertising the available methods.

The GET method is safe, meaning that applying it to a resource does not result in a state change of the resource (read-only semantics).[16]:§4.2.1 The GET, PUT, and DELETE methods are idempotent, meaning that applying them multiple times to a resource results in the same state change of the resource as applying them once, though the response might differ.[16]:§4.2.2 The GET and POST methods are cacheable, meaning that

responses to them are allowed to be stored for future reuse.[16]:§4.2.3

Discussion

Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs. This is because REST is an architectural style, while SOAP is a protocol. REST is not a standard in itself, but RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML. Many developers describe their APIs as being RESTful, even though these APIs do not fulfill all of the architectural constraints described above

(especially the uniform interface constraint).[15]

也可以看看

- Clean URL – URL intended to improve the usability of a website
- Content delivery network – Layer in the internet ecosystem addressing bottlenecks
- Domain Application Protocol (DAP)
- List of URI schemes – Namespace identifier assigned by IANA
- Microservices – Collection of loosely coupled services used to build computer applications

- Overview of RESTful API Description Languages
    - OpenAPI Specification – A specification for machine-readable interface files
    - Open Data Protocol – An open protocol for creating interoperable REST APIs
    - RAML
    - RESTful Service Description Language (RSDL)
- Resource-oriented architecture (ROA)
- Resource-oriented computing (ROC)
- Service-oriented architecture (SOA)
- Web-oriented architecture (WOA)

參考

1. *Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)" (http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.*
2. *Fielding, Roy (June 2014). "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Section 4" (https://tools.ietf.org/html/rfc7231#section-4) . IETF. Internet Engineering Task Force (IETF). RFC 7231 (https://tools.ietf.org/html/rfc7231) . Retrieved 2018-02-14.*

3. *Dusseault, Lisa (March 2010). "PATCH Method for HTTP" (https://tools.ietf.org/html/rfc5789) . IETF. Internet Engineering Task Force (IETF). RFC 5789 (https://tools.ietf.org/html/rfc5789) . Retrieved 2022-06-25.*
4. *"Fielding discussing the definition of the REST term" (https://web.archive.org/web/20151105014201/https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/6735) . groups.yahoo.com. Archived from the original (https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/6735) on November 5, 2015. Retrieved 2017-08-08.*

5. *Couldry, Nick (2012). Media, Society, World: Social Theory and Digital Media Practice (https://books.google.com/books?id=AcHvP9trbkAC&pg=PA2) . London: Polity Press. p. 2. ISBN 9780745639208.*
6. *Fielding, Roy Thomas (2000). "Chapter 6: Experience and Evaluation" (https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.*

7. *Erl, Thomas; Carlyle, Benjamin; Pautasso, Cesare; Balasubramanian, Raj (2012). "5.1". SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Upper Saddle River, New Jersey: Prentice Hall. ISBN 978-0-13-701251-0.*
8. *Fielding, Roy Thomas (2000). "Chapter 2: Network-based Application Architectures" (http://www.ics.uci.edu/~fielding/pubs/dissertation/net_app_arch.htm) . Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.*

9. *Richardson, Leonard; Ruby, Sam (2007). RESTful Web Services (https://archive.org/details/restfulwebservic00rich_0) . Sebastopol, California: O'Reilly Media. ISBN 978-0-596-52926-0.*
10. *Lange, Kenneth (2016). The Little Book on REST Services (https://www.kennethlange.com/the-little-book-on-rest-services/) . Copenhagen. p. 19. Retrieved 18 August 2019.*
11. *"REST HATEOAS" (http://restfulapi.net/hateoas/) . RESTfulAPI.net. 2 June 2018.*

12. *Ivan Salvadori, Frank Siqueira (June 2015). "A Maturity Model for Semantic RESTful Web APIs" (https://www.researchgate.net/publication/281287283) . Conference: Web Services (ICWS), 2015 IEEE International Conference OnAt: New York - USA – via Researchgate.*
13. *"What is REST API" (http://restfulapi.net/) . RESTful API Tutorial. Retrieved 29 September 2016.*
14. *Richardson, Leonard; Amundsen, Mike (2013), RESTful Web APIs, O'Reilly Media, ISBN 978-1-449-35806-8*
15. *Roy T. Fielding (2008-10-20). "REST APIs must be hypertext driven" (http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) . roy.gbiv.com. Retrieved 2016-07-06.*

16. *Fielding, Roy (June 2014). "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Section 4" (https://tools.ietf.org/html/rfc7231#section-4) . IETF. Internet Engineering Task Force (IETF). RFC 7231 (https://tools.ietf.org/html/rfc7231) . Retrieved 2018-02-14.*

延伸閱讀

- Pautasso, Cesare; Wilde, Erik; Alarcon, Rosa (2014), *REST: Advanced Research Topics and Practical Applications* (https://www.springer.com/engineering/signals/book/978-1-4614-9298-6) , Springer, ISBN 9781461492986
- Pautasso, Cesare; Zimmermann, Olaf; Leymann, Frank (April 2008), "RESTful

Web Services vs. Big Web Services: Making the Right Architectural Decision" (http://www.jopera.org/docs/publications/2008/restws) , *17th International World Wide Web Conference (WWW2008)*

- Ferreira, Otavio (Nov 2009), *Semantic Web Services: A RESTful Approach* (https://otaviofff.github.io/restful-grounding/) , IADIS, ISBN 978-972-8924-93-5
- Fowler, Martin (2010-03-18). "Richardson Maturity Model: steps towards the glory of REST" (https://martinfowler.com/articles/richardsonMaturityModel.html) . *martinfowler.com*. Retrieved 2017-06-26.