Web Services

Seminar

Name:Borade Prasad Sitaram

Class:M.sc(computer science)-1

Seminar Topic:RESTful Web Services

Abstract

A Web service is a Web page that is meant to be consumed by an autonomous program. Web service requires an architectural style to make sense of them as there need not be a human being on the receiver end to make sense of them.A Web service is a Web page that is meant to be consumed by an autonomous program. Web service requires an architectural style to make sense of them as there need not be a human being on the receiver end to make sense of them.

REST (REpresentational State Transfer) represents the model of how the modern Web should work. It is an architectural pattern that distills the way the Web already works. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

By its nature, user actions within a distributed hypermedia system require the transfer of large amounts of data from where the data is stored to where it is used. Thus, the Web architecture must be designed for large-grain data transfer. The architecture needs to minimize the latency as much as possible. It must be scalable, secure and capable of encapsulate legacy and new elements well, as Web is subjected to constant change. REST provides a set of architectural constraints that, when applied as a whole, address all above said issues.

Introduction of RESTful Web Services

REST is a hybrid architectural pattern which has been derived from the following network based architectural patterns.

Client-Server: Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

Stateless: Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. This constraint induces the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures. Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn’t have to manage resource usage across requests.

Cache: In order to improve network efficiency, we add cache constraints to form the client-cache stateless. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

Uniform Interface: The central feature that distinguishes the REST architectural style from other network based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide.

Layered System: The layered system style allows architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot see beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients.

Code on Demand: REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.

Working

GET: retrieve whatever information (in the form of an entity) is identified by the Request-URI. Retrieve representation, shouldn’t result in data modification.

POST: request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI. Annotation of existing resources, extending a database through an append operation, posting a message to a bulletin board, or group of articles. Used to change state at the server in a loosely coupled way (update).

PUT: requests that the enclosed entity be stored under the supplied Request-URI, create/put a new resource. Its used to set some piece of state on the server.

DELETE: requests that the origin server deletes the resource identified by the Request-URI.

REST provides a hybrid of all three options by focusing on a shared understanding of data types with metadata. REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient

and the nature of the resource. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface.

The benefits of the mobile object style are approximated by sending a representation that consists of instructions in the standard data format of an encapsulated rendering engine (e.g., Java). REST therefore gains the separation of concerns of the client-server style without the server scalability problem, allows information hiding through a generic interface to enable encapsulation and evolution of services, and provides for a diverse set of functionality through downloadable feature-engines.