RESTful Web Services

A
Seminar Report

by

Astik Anand
B130542CS

Department of Computer Science and Engineering
National Institute of Technology, Calicut

Monsoon-2016

National Institute of Technology, Calicut
Department of Computer Science and Engineering

Certified that this Seminar Report entitled

RESTful Web Services

is a bonafide record of the Seminar presented by

Astik Anand
B130542CS

in partial fullfillment of
the requirements for the award of the degree of
Bachelor in Technology in
Computer Science and Engineering

Bharath Narayanan
(Group Seminar Coordinator)
Department of Computer Science and Engineering

Acknowledgement

I have taken efforts in this seminar. However, it would not have been possible without the kind support and help of many individuals and organisation. I would like to extend my sincere thanks to all of them.

I am highly indebted to (Mr. Bharath Narayanan) for his guidance and supervision.

Abstract

A Web service is a Web page that is meant to be consumed by an autonomous program and 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 theme.
REST(Representational State Transfer) represents the model of how the modern Web should work. It is an architectural pattern that distils 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.
RESTful Web services play important role in distributed web applications. Performance of Web services affects overall performance of applications.
Using features of REST, HTTP, and the REST frameworks like Jersey, Restlet, RESTEasy the latency and system resource consumption of application is improved. Performance of RESTful web services is improved using techniques such as fast manipulation of strings,streaming large representations, compressing SOAP response, partial representation, using Caching techniques and using conditional methods. It is observed that performances of RESTful Web services increases by approximately 40% if above techniques are used.

Table of Contents

1. Introduction

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.

2. Deriving REST

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.

3. The Elements of REST

- **Components:** Communicate by transferring representations of resources through a standard interface rather than operating directly upon the resource itself. They provide or mediate access to the resource.
  Ex: Origin Servers, Gateways, Proxies, User Agents.
- **Connectors:** Hide the implementation details of the communication mechanism from the components by providing abstract interface.
  Ex: Servers, Resolvers, Tunnels, Caches.

- **Data Elements:** Following table lists different REST data elements.

Data Elements	Example
Resource	Image or any required information
Resource Identifier	URL
Representation	JPEG Image, Plaintext document
Representation Metadata	Media-type, Last Modified time
Resource Metadata	Source Links
Control Data	Cache

3.1 Resource and Resource Identifier

Resource is basically a concept. Any raw information that might be the target of hypertext reference can be termed as a resource. Ex: an image or whether detail or even non-virtual object like a person can be a resource. Resource can be static i.e. they correspond to same value set even when referred at a time after creation or they may vary constantly. The abstract definition of the resource has the advantage of referring the concept rather than the representation, thus removing the need to change all the links when the representation changes. Resources are decoupled from their representation.

REST uses a *resource identifier* to identify the particular resource involved in an interaction between components. It relies instead on the author choosing a resource identifier that best fits the nature of the concept being identified.

From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate. A third RESTful Web service characteristic is all about the URIs.

REST Web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and super-ordinate branches connected at nodes.

Some points to make note while structuring the URI for restful web services:

- Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so that the port can be changed to something else without changing the URIs.
- Everything should be in lower case.
- Substitute spaces with hyphens or underscores.
- Instead of using the 404 Not Found code if the request URI is for a partial path, always provide a default page or resource as a response.

3.2 Representation

A *representation* is a sequence of bytes, plus *representation metadata* to describe those bytes. The web services work on the resources using representation. A representation consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for the purpose of verifying message integrity).

3.3 Control Data

It defines the purpose of a message between components, such as the action being requested or the meaning of a response. The recipient can process the message using the control data and the nature of the media type of the representation.

4. Disadvantages of Non-Restful Methods

- They don't scale well
- They have significantly higher coordination costs.

5. Working

Following operations are used:

- 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.

The following table summarizes the REST principles

CRUD	REST	
Create	Put	Initialize the state of a new resource at the given URI
Read	Get	Retrieve the current state of the resource
Update	Post	Modify the state of the resource
Delete	Delete	Clear the resource when it is no longer valid.

A distributed hypermedia architect has only three fundamental options:

- Render the data where it is located and send a fixed-format image to the recipient.
- Encapsulate the data with a rendering engine and send both to the recipient.
- Send the raw data to the recipient along with metadata that describes the data type, so that the recipient can choose their own rendering engine.

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.

6. REST features

- Resources are manipulated through their representations.
- Messages are self-descriptive and *stateless.*
- Multiple representations are accepted or sent.
- Hypertext is the engine of application state.

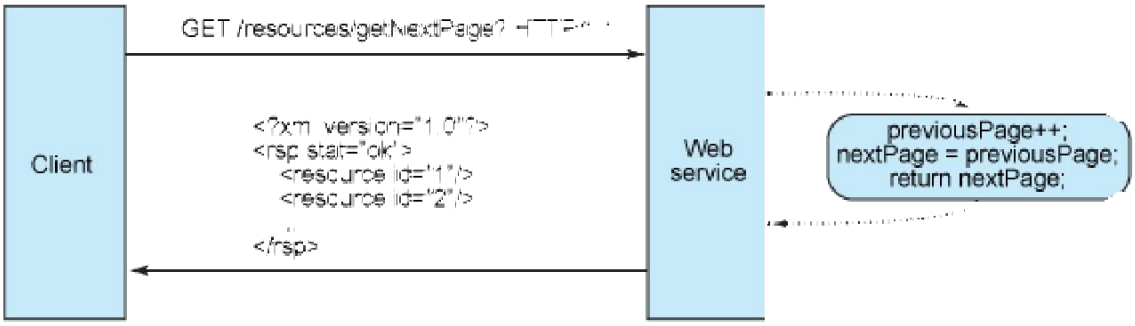
“State” means application/session state. Application state is the information necessary to understand the context of an interaction. Authorization and authentication information are examples of application state. The "stateless" constraint means that all messages must include all application state. It is maintained as part of the content transferred from client to server back to client. Thus any server can potentially continue transaction from the point where it was left off. User can pick up where you left off by merely accessing the URI at a later time, regardless of client or server changes.

7. REST is STATELESS

REST Web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a Web service call. Using intermediary servers to improve scale requires REST Web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

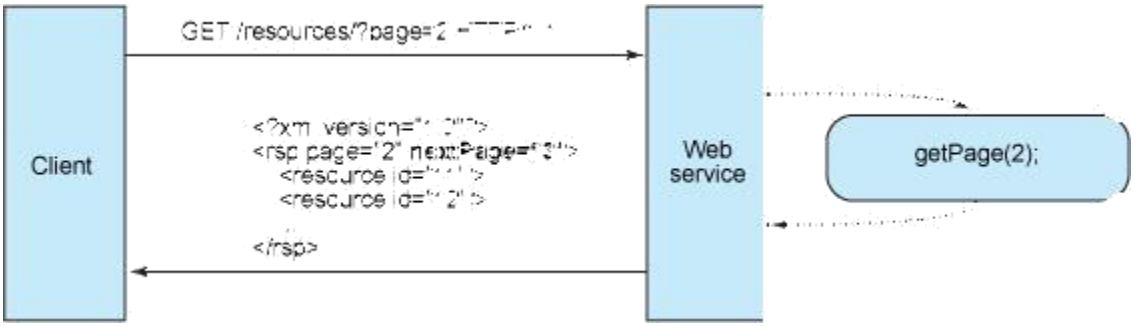
A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST Web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Following illustrates a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a previousPage variable somewhere to be able to respond to requests for next.



Stateful Design

Stateful services like this get complicated. Stateless server-side components, on the other hand, are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful Web service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for next.



Stateless Design

A stateless Web service generates a response that links to the next page number in the set and lets the client do what it needs to in order to keep this value around. This aspect of RESTful Web service design can be broken down into two sets of responsibilities as a high-level separation that clarifies just how a stateless service can be maintained:

Server

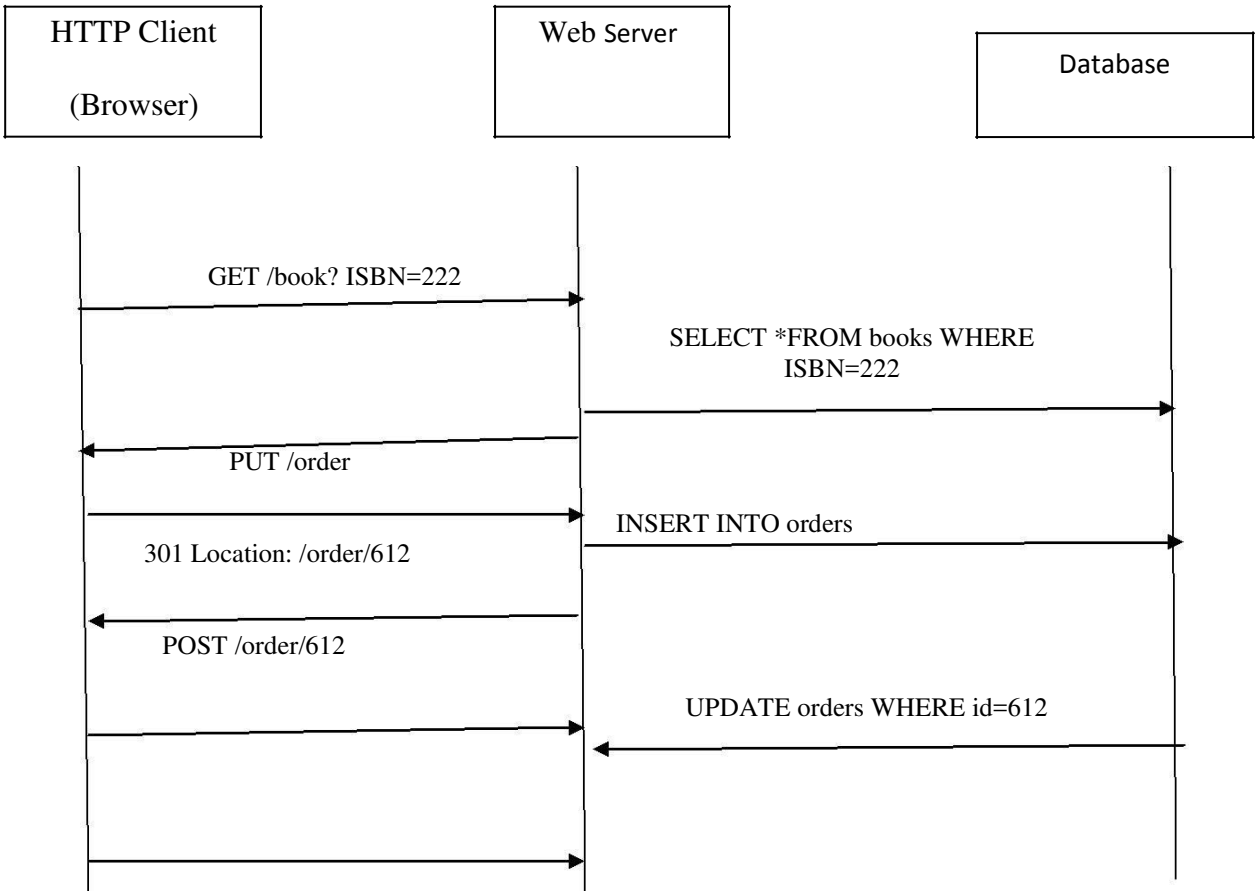
- Generates responses that include links to other resources to allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources so that these remain connected.
- Generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a Cache-Control and Last-Modified (a date value) HTTP response header.

Client application

- Uses the Cache-Control response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the Last-Modified response header and sends back the date value in an If-Modified-Since header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand in hand in that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes.
- Sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the Web service interface and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server, the server's ability to add context to a request, or about application state that is kept in between requests.

This collaboration between client application and service is essential to being stateless in a RESTful Web service. It improves performance by saving bandwidth and minimizing server-side application state.

8. RESTful Web application Example



9. Advantages:

- Scalable component interactions
- General interfaces
- Independently deployed connectors.
- Reduced interaction latency.
- Strengthened security.
- Safe encapsulation of legacy systems.
- Supports intermediaries (proxies and gateways) as data transformation and caching components.
- Separates server implementation from the client's perception of resources ("Cool URIs Don't Change").
- Scales well to large numbers of clients.
- Enables transfer of data in streams of unlimited size and type.

Disadvantages:

- It sacrifices some of the advantages of other architectures.
- Stateful interaction with an FTP site.
- It retains a single interface for everything
- The stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context. In addition, placing the application state on the client-side reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of semantics across multiple client versions.

10. Conclusion:

Service-Oriented Architecture can be implemented in different ways. General focus is on whatever architecture gets the job done. SOAP and REST have their strengths and weaknesses and will be highly suitable to some applications and positively terrible for others. The decision of which to use depends entirely on the circumstances of the application.

11. References:

[1]. Thomas Bayer, REST Web Services – Eine Einfuehrung (November 2002)

[2]. International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)
Volume 5, Issue 11, November 2015

[3]. IOSR Journal of Computer Science (IOSR-JCE) e-ISSN: 2278-0661, p-ISSN: 2278-8727 PP 12-16