Restful Web Services

Kunal Agrawal[1] Prof. Mr. S. S. Agrawal[2]
[1,2]College of Engineering & Technology, Akola, India

Abstract— Web's major goal was to be a shared data house through which individuals and machines may communicate. By its nature, user actions at intervals a distributed multimedia system need the transfer of enormous amounts of information from wherever the info is hold on to wherever it's used. Thus, the net design should be designed for large-grain information transfer. The design must minimize the latency the maximum amount as attainable. REST is employed to create net services that ar light-weight, rectifiable, and climbable in nature. A service that is made on the remainder design is termed a reposeful service. The underlying protocol for REST is communications protocol, that is that the basic net protocol. REST stands for representational State transfer. REST may be a thanks to access resources that consist a specific atmosphere. For instance, you'll have a server that might be hosting necessary documents or photos or videos. All of those ar Associate in Nursing example of resources. If a consumer, say an internet browser wants any of those resources, it's to send a call for participation to the server to access these resources. currently REST defines the way on however these resources are often accessed. It should be climbable, secure and capable of encapsulate bequest and new parts well, as net is subjected to constant modification. REST provides a collection of beaux arts constraints that, once applied as an entire, address all higher than aforementioned problems.

Keywords: Restful Web Services

I. 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 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.

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 behaviour 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 legacyclients. 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.

II. Literature Survey

A. Background History

REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use.

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 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.

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. [2,4]

Reliability is improved because it eases the task of recovering from partial failures. Scalability is improved because not having to store state between requests 4 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 behaviour 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.

B. Problems in Existing System

SOAP is a protocol which was designed before REST and came into the picture. The main idea behind designing SOAP was to ensure that programs built on different platforms and programming languages could exchange data in an easy manner. SOAP stands for Simple Object Access Protocol. SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern. SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.

HTTP is a synchronous, request/response protocol. This means the protocol does not inherently support server-initiated notifications (peer-to-peer), which are often required. That's why call-backs in Retain applications require the use of application-level design patterns like Webhooks. Now that we have a bidirectional transport protocol in the form of WebSocket's, perhaps the industry should be looking at layering a new application protocol on top of it that follows Retain principles." This is interesting, given that over the past year or so we have seen others discussed whether WebSocket's and REST are even compatible. There is at least something that REST community could learn from the Web Services stack: "These are all end-to-end protocols layered on top of the core SOAP+WS-Addressing "messaging" capability." Others have suggested similar in the past. Ganesh then goes on to discuss an analogy between the way Web Services uses WS-Addressing, WS-Reliable Messaging, WS-Secure Conversation and WS-Policy and the internet equivalents including TCP, IP and IPsec. Ganesh suggests that REST's application idempotence may be better for reliability (though does not define better in any context), and perhaps there are alternatives to transactions for REST, but he is left with:

HTTP has too few verbs, particularly if you want to do peer-to-peer interactions and has a few suggestions: "INCLUDE (add to a resource collection and return a server-determined URI), PLACE (add to a resource collection with client-specified URI),REPLACE (in toto), FORCE (PLACE or REPLACE), AMEND (partial update, a container verb specifying one or more other verbs to specify operations on a resource subset), MERGE (populate parts of the resource with the supplied representation),RETIRE (a better word than DELETE) and SOLICIT (a GET replacement that is also a container verb, to tell the responding peer what to do to the initiator's own resource(s), because this is a peer-to-peer world now)

HTTP combines application-level and transport-level status codes (e.g., 304 Not Modified and 400 Bad Request vs 407 Proxy Authentication Required and 502 Bad Gateway). The next implementation of REST on another transport should design for a cleaner separation between the application protocol and the transport protocol. HTTP does double-duty and the results are often a trifle inelegant.

C. 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. [5,8]

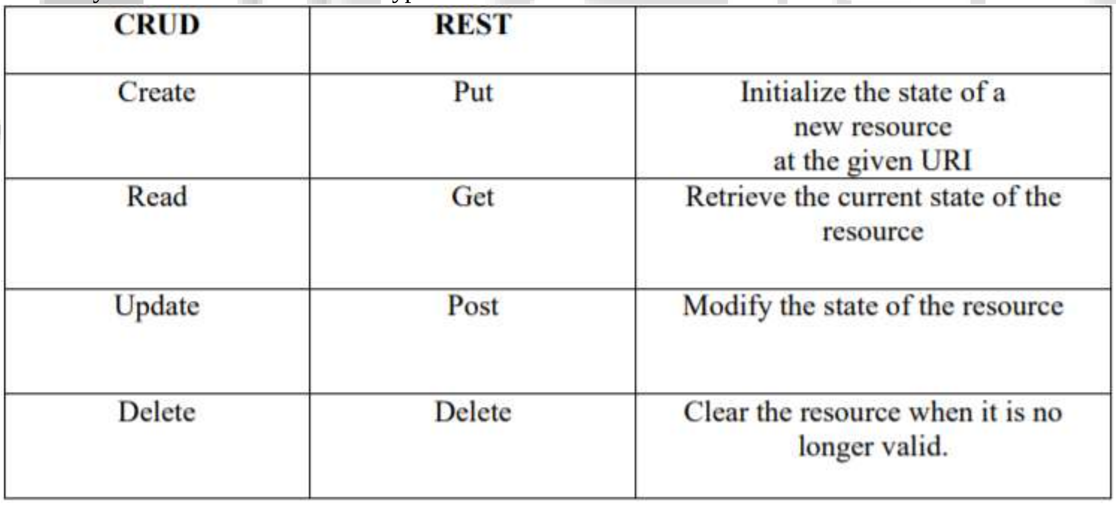
From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate. A third RESTful Web service characteristic is all about the URIs. REST Web service 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 sub paths 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. [1,2]

Some points to make note while structuring the URI for restful web services: Hide the server-side scripting technology file extensions (. asp, pup, .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.[6]

D. 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. It’s used to set some piece of state on the server.
– DELETE: requests that the origin server deletes the resource identified by the Request URI.



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.

Fig. 2.1: This table summarizes the REST principles

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 functionalities through downloadable feature-engines.

III. Related Work

A Web Service is any service that is available over the Internet, uses a standardized XML messaging system and is not tied to any one operating system or programming language. A Web Service is seen as an application accessible to other applications over the web. However, despite the commonness of the two definitions, they are still too rough for practical usage. In a broader sense of the word, anything that has an URL can be considered as a Web Services.

A more precise definition is provided by the UDDI consortium, which characterizes Web services as "self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. This definition is more detailed, placing the emphasis on the need for being compliant with Internet standards. In addition, it requires the service to be open, which essentially means that it has a published interface that can be invoked across the Internet. Inspire of this clarification, the definition is still not precise enough. It is not clear what it is meant by a modular, self-contained business application. A more detailed definition is provided by the World Wide Web consortium (W3C), the group involved in the Web Service Activity. Web service is" A software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifices. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols". [2,5]

The W3C definition is quite accurate and also hints at how Web Service should work. The definition stresses that Web services should be capable of being "defined, described and discovered," thereby clarifying the meaning of "accessible" and making more concrete the notion of "Internet-oriented, standards-based interfaces." It also states that Web services are similar to the services in conventional middleware. Web services are components that can be integrated into more complex distributed applications. The W3C also states that XML is part of the solution. More specific definitions exist in Literature. In the online technical dictionary Zebedia, a Web service is defined as "a standardized way of integrating Web based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available, and UDDI is used for listing what services are available". [4,8]

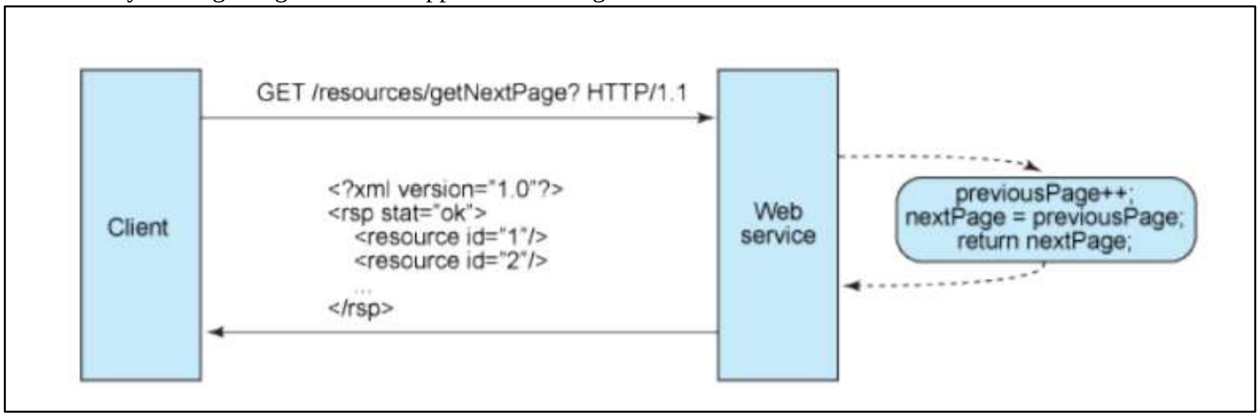
Specific standards that could be used for performing binding and for interacting with a web service are mentioned in the above definition. These are the leading standards today in Web services. Many applications that are "made accessible to other applications" do so through SOAP, WSDL, UDDI and other Web standards. However, these standards do not constitute the essence of Web services technology.[10]

IV. System design & implementation

A. *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 previous Page variable somewhere to be able to respond to requests for next. [3,6]



```
<?xml version="1.0"?>
<rsp stat="ok">
   <resource id="1"/>
   <resource id="2"/>
   ...
</rsp>
```

Fig. 4.1: 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.[7]

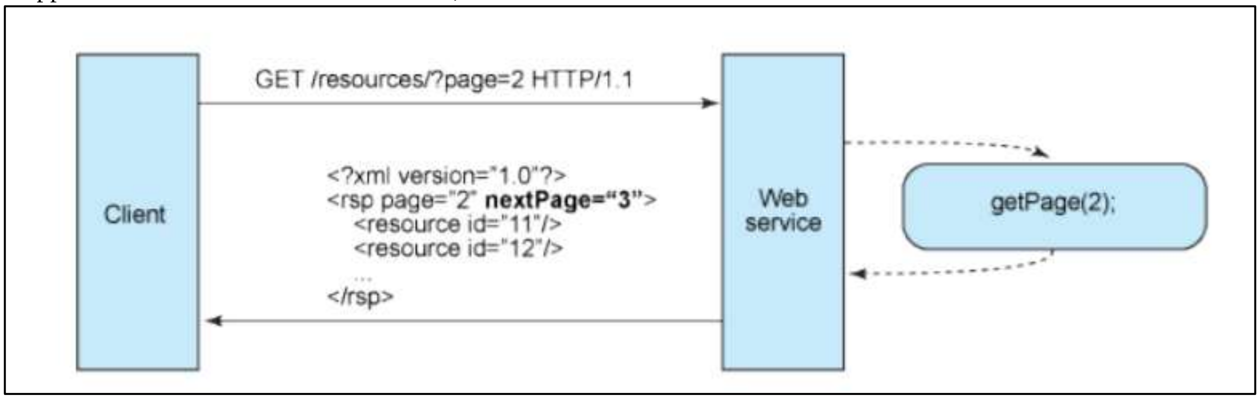


Fig. 4.2: Stateless Design

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:

B. *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.

C. *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.

V. MERITES & DEMERITES

A. *Merits*

- The restful web services have the scalable equipment interactions.
- It possesses the general interfaces.
- It can independently deploy the connections.
- It decreases the interaction latency.
- It strengthens or improves the security.
- It has the feature of safe encapsulation of legacy systems.
- It sustains the proxies and gateways as information transformation and caching equipment.
- Restful web services scale to a huge number of clients.
- It enables the alienate of information in streams that possess infinite size.
- Scalable component interactions.
- General interfaces.
- Independently deployed connectors.
- Reduced interaction latency.
- 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.

B. *Demerits*

- It destroys few advantages of other architectures.
- Restful web services have a state of interaction with an FTP site.

– It contains a single interface for everything.
– It reduces the performances of the new by enhancing the repetitive information.
– 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 behaviour, since the application becomes dependent on the correct implementation of semantics across multiple client versions.

VI. Application/comparative study

RESTFUL WEB APPLICATION EXAMPLE

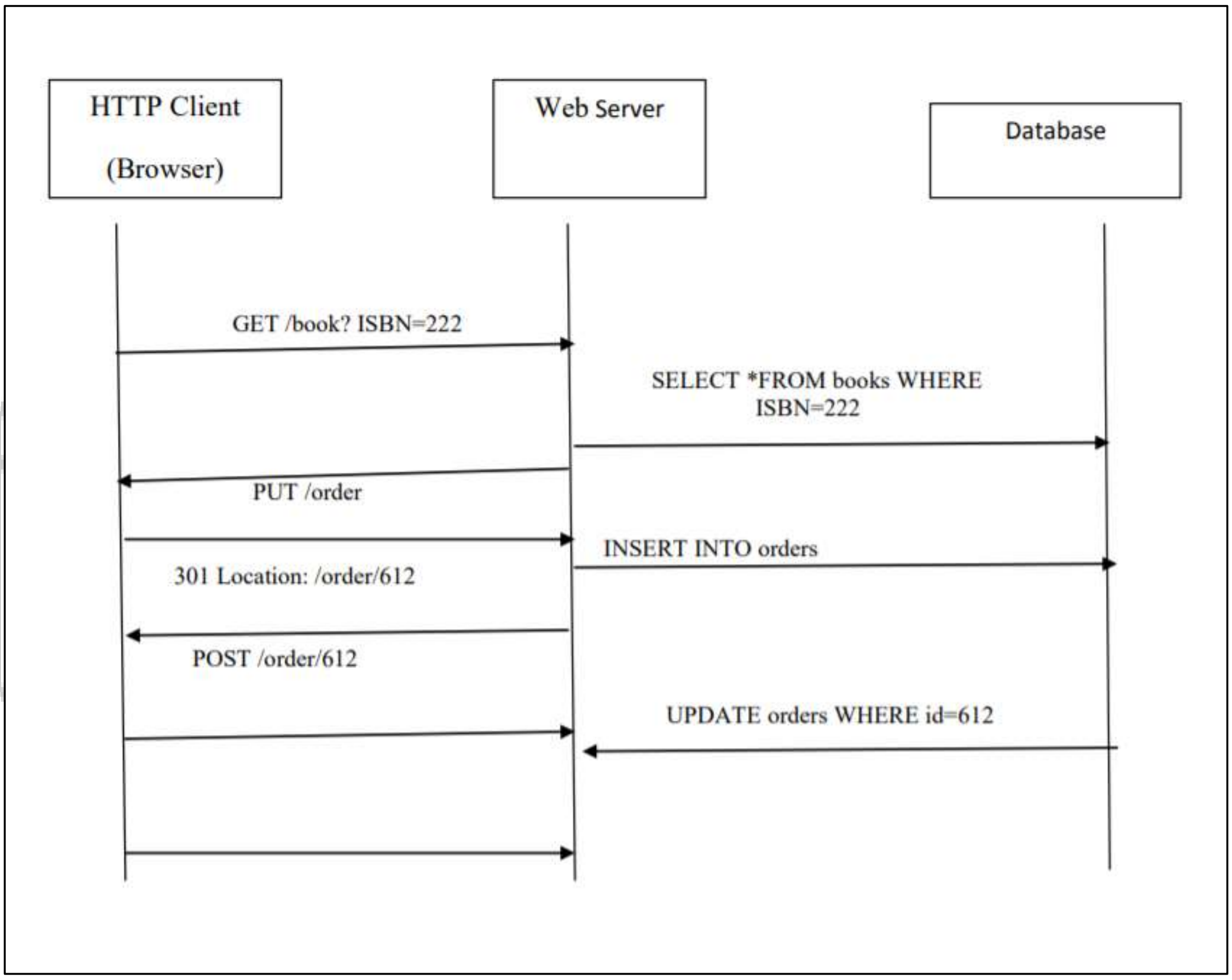


Fig. 5.1: Restful Web application Example

VII. 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.
Constructing application to application Web services is simple, in principle. SOAP, WSDL, and UDDI are:
– Unavoidable framework for XML communication and service description.
– Surprisingly complicated and their benefits are not always obvious, although the basic ideas behind them are great.
– Still under development.

REST stands for Representational State Transfer. REST is used to build web services that are lightweight, maintainable, and scalable in nature. More and more applications are moving to the Restful architecture. This is because there are a lot of people now using mobile devices and a wider variety of applications moving to the cloud. The main aspects of REST are the resources which reside on the server and the verbs of GET, POST, PUT and DELETE, which can be used to work with these resources. Visual Studio and.Net can be used to create Restful web services. When Testing web services for POST and PUT, you need to use another tool called fiddler which can be used to send the POST and PUT request to the server.

REFERENCES

[1] T. Berners-Lee., “WWW: Past, present, and future”. IEEE Computer, 29(10), Oct. 1996, pp. 69–77.
[2] Petasos, Cesare; Wilde, Erik; Alarcon, Rosa (2014), REST: Advanced Research Topics and Practical Applications, Springer, ISBN 9781461492986
[3] Petasos, Cesare; Zimmermann, Olaf; Lemann, Frank (April 2008), "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision", 17th International World Wide Web Conference (WWW2008)
[4] Ferreira, Octavio (Nov 2009), Semantic Web Services: A RESTful Approach, IADIS, ISBN 978-972-8924-93-5
[5] Fowler, Martin (2010-03-18). "Richardson Maturity Model: steps towards the glory of REST". Martinfowler.com. Retrieved 2017-06-26.
[6] Thomas Bayer, “REST Web Services” – Eine Einfühlung (November 2002)
[7] Little, Mark (October 1, 2008). "A Comparison of JAX-RS Implementations".
[8] Hadley, Marc and Paul Sandoz, eds. (September 17, 2009). JAX-RS: Java API for “RESTful Webservices “.
[9] Investigating Web Services on the World Wide Web, the analysis presented at the, “WWW 2008 conference”.
[10] J. Elson, L. Giroud, and D. Destrin. Fine-grained network time synchronization using reference broadcasts. In Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, USA., dec 2002.