A

Seminar report

on

RESTful Web services

Submitted in partial fulfillment of the requirement for the award of degree
Of Computer Science

SUBMITTED TO: **SUBMITTED BY:**

www.studymafia.org www.studymafia.org

Preface

I have made this report file on the topic **RESTful Web services** ; I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to …………..who assisting me throughout the prepration of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

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. 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

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 late ncy, enforce security, and Encapsulate legacy systems.

What is REST?

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.

REST Web Services Characteristics

Here are the characteristics of REST:

- Client-Server: a pull-based interaction style: consuming components pull representations.
- Stateless: each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- Cache: to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- Uniform interface: all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- Named resources - the system is comprised of resources which are named using a URL.
- Interconnected resource representations - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.
- Layered components - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

Principles of REST Web Service Design

1. The key to creating Web Services in a REST network (i.e., the Web) is to identify all of the conceptual entities that you wish to expose as services. Above we saw some examples of resources: parts list, detailed part data, purchase order.

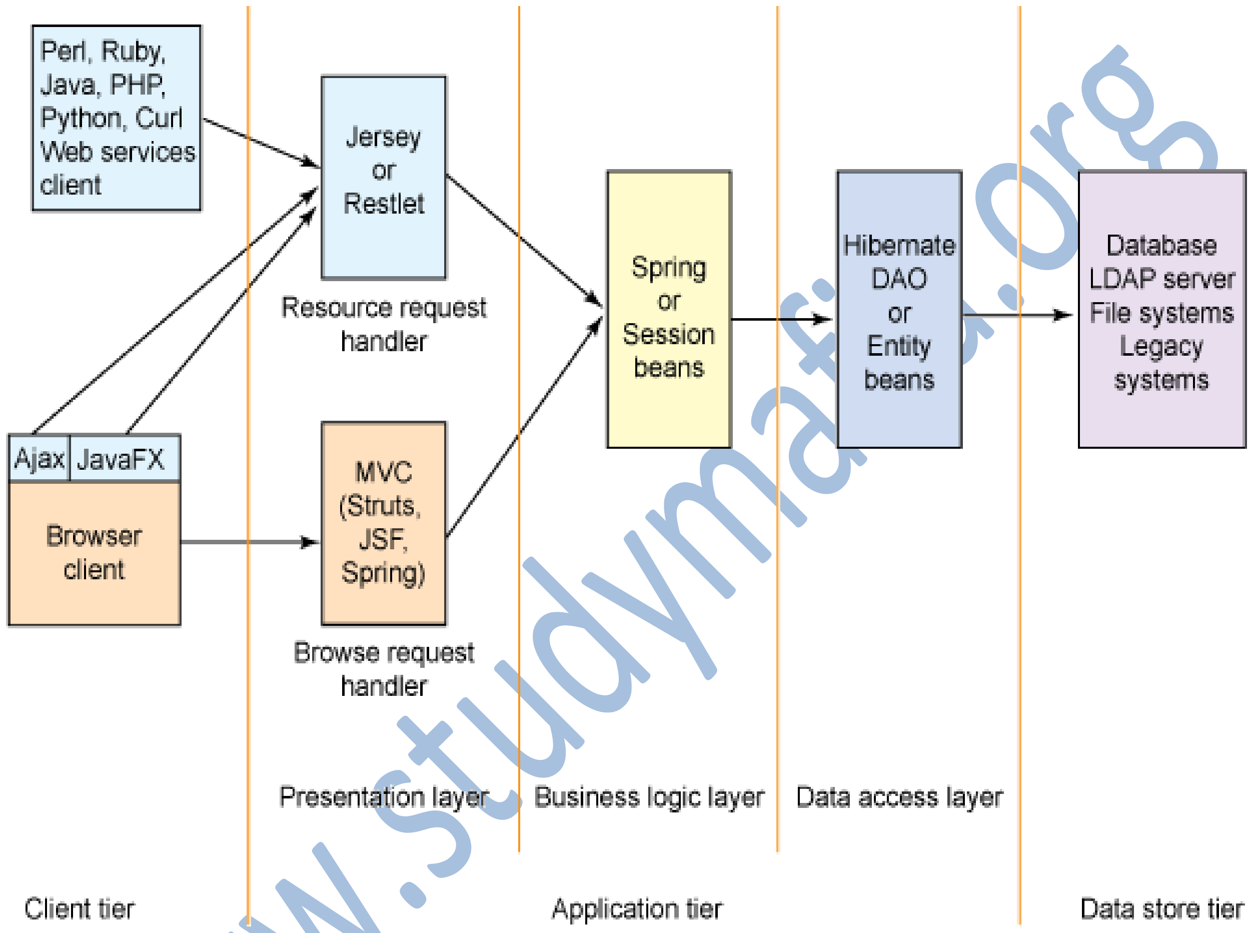
2. Create a URL to each resource. The resources should be nouns, not verbs. For example, do not use this:

http://www.parts-depot.com/parts/getPart?id=00345

Note the verb, getPart. Instead, use a noun:
http://www.parts-depot.com/parts/00345

3. Categorize your resources according to whether clients can just receive a representation of the resource, or whether clients can modify (add to) the resource. For the former, make those resources accessible using an HTTP GET. For the later, make those resources accessible using HTTP POST, PUT, and/or DELETE.
4. All resources accessible via HTTP GET should be side-effect free. That is, the resource should just return a representation of the resource. Invoking the resource should not result in modifying the resource.
5. No man/woman is an island. Likewise, no representation should be an island. In other words, put hyperlinks within resource representations to enable clients to drill down for more information, and/or to obtain related information.
6. Design to reveal data gradually. Don't reveal everything in a single response document. Provide hyperlinks to obtain more details.
7. Specify the format of response data using a schema (DTD, W3C Schema, RelaxNG, or Schematron). For those services that require a POST or PUT to it, also provide a schema to specify the format of the response.
8. Describe how your services are to be invoked using either a WSDL document, or simply an HTML document.

RESTful Web Services Architecture



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.

REST Design Guidelines

Some soft guidelines for designing a REST architecture:

1. Do not use "physical" URLs. A physical URL points at something physical -- e.g., an XML file: "http://www.acme.com/inventory/product003.xml". A *logical* URL does not imply a physical file: "http://www.acme.com/inventory/product/003".
    - Sure, even with the .xml extension, the content could be dynamically generated. But it should be "humanly visible" that the URL is logical and not physical.
2. Queries should not return an overload of data. If needed, provide a paging mechanism. For example, a "product list" GET request should return the first *n* products (e.g., the first 10), with next/prev links.
3. Even though the REST response can be anything, make sure it's well documented, and do not change the output format lightly (since it will break existing clients).
    - Remember, even if the output is human-readable, your clients aren't human users.
    - If the output is in XML, make sure you document it with a schema or a DTD.
4. Rather than letting clients construct URLs for additional actions, include the actual URLs with REST responses. For example, a "product list" request could return an ID per product, and the specification says that you should use http://www.acme.com/product/*PRODUCT_ID* to get additional details. That's bad design. Rather, the response should include the actual URL with each item: http://www.acme.com/product/001263, etc.
    - Yes, this means that the output is larger. But it also means that you can easily direct clients to new URLs as needed, without requiring a change in client code.
5. GET access requests should never cause a state change. Anything that changes the server state should be a POST request (or other HTTP verbs, such as DELETE).

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.