RESTful Service In Java

REST, What is it actually?

- REST Stands for REpresentational State Transfer
- It is an architectural style, and an approach to communications that is often used in the development and access of Web services.
- The term ***representational state transfer*** was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at **UC Irvine**.
- Systems that conform to the constraints of REST they can be called RESTful.
- RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP)
  - They use the same HTTP verbs (GET, POST, PUT, DELETE, etc.)

REST Architectural constraints

- The formal REST constraints are
  - Client–server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)
  - Uniform interface

REST Architectural constraints – Client Server

- A uniform interface separates clients from servers.
- Clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved.
- Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable.
- Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

REST Architectural constraints – **Stateless**

- They are stateless
- no client context being stored on the server between requests.
- Each request from any client contains all the information necessary to service the request, and session state is held in the client.

REST Architectural constraints – **Cacheable**

- As on the World Wide Web, clients and intermediaries can cache responses.
- Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing 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.

REST Architectural constraints – **Cacheable**

- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches.
- They may also enforce security policies.

REST Architectural constraints – **Code on demand (optional)**

- Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.
- Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

REST Architectural constraints – **Uniform interface**

- The Uniform Interface is a constraint that is placed on REST services in order to simplify things (and ensure that services can be managed independently from one another) and they are as follows :

- **Identification Of Resources**
  This basically states that a request will need to identify the resources that it is looking for (via a URL). Additionally, the resources themselves may not have any relationship with how they are returned to the client (i.e. you can request a given resource in JSON, XML, or some other format based on how your API was built).
- **Resource Manipulation through Representation**
  This basically states that when a client has a given resource, along with any metadata, that they should have enough information to either modify or delete the resource (i.e. there isn't anything left out that it would need to call to the API to do these things).

REST Architectural constraints – **Uniform interface**

- **Self Descriptive Messages**
  A message should have enough information to let the server know how to process it (i.e. the type of request, mime types, etc.)
- **Hypermedia as the Engine of Application State (HATEOAS)**
  Accessing an API should be similar to accessing a web page (i.e. you should be able to discover other areas of the API much as a user would discover links on a page). Basically a response can contain links and point to other areas of the API that are available.

How Do RESTful applications Work?

- RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data.
  - Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.
- REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al)
  - Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture.
- REST is not a "standard".
  - There will never be a W3C recommendation for REST, for example.
  - And while there are REST programming frameworks, working with REST is so simple that you can often "roll your own" with standard library features in languages like Perl, Java, or C#.

REST As Web Service

- As a programming approach, REST is a lightweight alternative to Web Services and RPC.
- Much like Web Services, a REST service is:
  - Platform-independent (you don't care if the server is Unix, the client is a Mac, or anything else),
  - Language-independent (C# can talk to Java, etc.),
  - Standards-based (runs on top of HTTP), and
  - Can easily be used in the presence of firewalls.

REST As Web Service

- Like Web Services, REST offers no built-in security features, encryption, session management, QoS guarantees, etc.
  - But also as with Web Services, these can be added by building on top of HTTP:
  - For security, username/password tokens are often used.
  - For encryption, REST can be used on top of HTTPS (secure sockets).

- One thing that is *not* part of a good REST design is cookies: The "ST" in "RE**ST**" stands for "State Transfer", and indeed, in a good REST design operations are self-contained, and each request carries with it (transfers) all the information (state) that the server needs in order to complete it.

REST As Web Service

- Web service APIs that adhere to the REST architectural constraints are called RESTful APIs.
- HTTP-based RESTful APIs are defined with the following aspects:
  - **base URL**, such as http://example.com/resources/
  - **an Internet media type** type that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json) The current representation tells the client how to compose all transitions to the next application state. This could be as simple as a URL or as complex as a java applet.
  - **standard HTTP methods** (e.g., OPTIONS, GET, PUT, POST, and DELETE)