RESTful Services

Naveen Kumar K S

Web-services

- Definition:
  - *“A Web service is a **method of communication** between two electronic devices **over the World Wide Web**.”*

    Wikipedia
  - *Web service is "a software system designed to support **interoperable** machine-to-machine interaction over a network.“*

    W3C

Web-services

- Definition:
  - A Web service has an interface described in a machine-processable format (specifically Web Services Description Language, aka **WSDL**).
  - Other systems interact with the Web service in a manner prescribed by its description using **SOAP** messages, typically conveyed using **HTTP** with an **XML serialization** in conjunction with other Web-related standards.

REST

- Representational State Transfer
- It is neither a framework nor a technology
  - It is an Architectural Style built on certain principles using the current HTTP web fundamentals
- It was first described by Roy Fielding in 2000
- **It's a resource-oriented Web-service**

REST

- The REST style is a group of six major constraints that induce the various properties needed for the Web
- In order for architecture to be viewed as REST architecture, it must satisfy these constraints

REST Constraints

- REST constraints are design rules that are applied to establish the distinct characteristics of the REST architectural style
- If you follow all constraints designed by the REST architectural style your systems is considered RESTful

REST Constraints

- These constraints don't dictate what kind of technology to use
  - They define how data is transferred between components and what benefits we get by following the guidelines
- Therefore, a RESTful system can be implemented in any networking architecture available

REST Constraints

- Client-Server
- Stateless
- Cacheable
- Uniform Interface
- Layered System
- Code On Demand (Optional)

REST Web-services

- ***Architecture definition***
  - In a REST based architecture everything is a resource. A resource is **accessed via a common interface based on the HTTP standard methods**.
  - In a REST based architecture you typically have
    - a **REST server** which **provides access** to the resources
    - and a **REST client** which **accesses and modify** the REST resources.
  - REST is designed to **use a stateless communication protocol**, typically HTTP.
  - REST allows that **resources have different representations**, e.g. text, xml, json etc. The rest client can ask for specific representation via the HTTP protocol (content negotiation).

REST Web-services

- ***HTTP methods***
  - GET defines a reading access of the resource without side-effects. The resource is never changed via a GET request, e.g. the request has no side effects (idempotent).
  - POST creates a new resource, must also be idempotent.
  - DELETE removes the resources. The operations are idempotent, they can get repeated without leading to different results.
  - PUT updates an existing resource or creates a new resource.

REST Web-services

- ***Example***
  - base URI for the web service: *http://example.com/resources/*

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://example.com/resources/	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as http://example.com/resources/item17	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

JAX-RS

- **JSR-311:** “JAX-RS: The Java API for RESTful Web Services”
- Introduced in Java SE 5: annotation-based
- Objective: simplify the development and deployment of web service clients and endpoints.
- Part of J2EE 6
- No configuration is necessary to start using JAX-RS

JAX-RS API

- **Specifications**
  - Annotations to aid in mapping a resource class (a Plain Old Java Object) as a web resource.
    - **@Path** specifies the relative path for a resource class or method.
    - **@GET**, **@PUT**, **@POST**, **@DELETE** and @HEAD specify the HTTP request type of a resource.
    - **@Produces** specifies the response MIME media types.
    - **@Consumes** specifies the accepted request media types.
  - Annotations to method parameters to pull information out of the request. All the @*Param annotations take a

    key of some form which is used to look up the value required.
    - **@PathParam** binds the parameter to a path segment.
    - **@QueryParam** binds the parameter to the value of an HTTP query parameter.
    - @MatrixParam binds the parameter to the value of an HTTP matrix parameter.
    - @HeaderParam binds the parameter to an HTTP header value.
    - @CookieParam binds the parameter to a cookie value.
    - @FormParam binds the parameter to a form value.
    - **@DefaultValue** specifies a default value for the above bindings when the key is not found.
    - @Context returns the entire context of the object. Ex.(@Context HttpServletRequest request)

JAX-RS implementations

- **Implementations**
  - Apache, CXF from Apache
  - **Jersey, from Sun/Oracle (J2EE 6 reference implementation)**
  - RESTeasy, from Jboss
  - Restlet, created by Jerome Louvel, a pioneer in REST frameworks.
  - …

REST CONSTRAINTS

Detailed explanation

Client-Server

- This is a major constraint and it simply requires the existence of a client component that sends requests and a server component that receives requests
- After receiving a request the server may also issue a response to the client
- This constraint is based on the principle of Separation of concerns

Client-Server

- Applying separation of concerns
  - Separates user interface concerns from data storage concerns
  - Improves portability of interface across multiple platforms
  - Improves scalability by simplifying server components
  - Allows the components to evolve independently

Stateless

- The notion of statelessness is defined from the perspective of the server
- The constraint says that the server should not remember the state of the application
  - As a consequence, the client should send all information necessary for execution along with each request, because the server cannot reuse information from previous requests as it didn’t memorize them
- All info needed is in message

Stateless

- By applying statelessness constraint
  - Session state is kept entirely on the client
  - Visibility is improved since a monitoring system does not have to look beyond a single request
  - Reliability is improved due to easier recoverability from partial failures
  - Scalability is improved due to not having to allocate resources for storing state
- Server does not have to manage resource usage across requests
- Stateless constraint has following tradeoffs:
  - Reduced Network Performance
  - Reduced server control over application consistency

Cacheable

- REST includes cache constraints so that the "second fetch" doesn't have to be made at all if the data is already sitting in your local cache (or can be a request that only uses partial network resources if the cache is an intermediary or server-side)
- If your data can be designed in such a way to take advantage of this, you can reduce total network traffic by orders of magnitude

Cacheable

- By adding optional non-shared caching:
  - Data within a response to a request is implicitly or explicitly labeled as cacheable or non cacheable
  - If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests
  - Improves efficiency, scalability and user perceived performance
- Tradeoff:
  - cacheable constraint reduces Reliability

Uniform Interface

- The uniform interface constraint, at a high level means the interface for a component needs to be generic as possible
- It simplifies and decouples the architecture, which enables each part of the architecture to evolve independently

Uniform Interface

- By applying uniform interface constraint:
  - Overall system architecture is simplified and the visibility of interactions is improved
  - Implementations are decoupled from the services they provide and encourage independent evolvability
- Trade off:
  - Degrades efficiency since information is transferred in a standardized form rather than one which is specific to application's needs

Layered System

- There are many layers between the client and the server
- These are called intermediaries and can be added at various points in the request-response path without changing the interfaces between components, where they do things to passing messages such as translation or improving performance with caching

- Intermediaries include proxies and gateways
- Proxies are chosen by the client, while gateways are chosen by the origin server or are imposed by the network

Layered System

- By applying a layered system constraint:
  - Similar to client-server constraint this constraint improves simplicity by separating concerns
  - Can be used to encapsulate legacy services or protect new services from legacy clients
  - Intermediaries can be used to improve system scalability by enabling load balancing
  - Placing shared caches at boundaries of organizational domain can result in significant benefits. Can also enforce security policies e.g. firewall
  - Intermediaries can actively transform message content since messages are self descriptive and their semantics are visible to the intermediaries
- Trade off:
  - Adds overhead and latency and reduce user perceived performance.

Code on Demand

- Code on demand is an optional constraint
- It allows a client to download and execute code from a server

Code on Demand

- By applying Code on demand constraint:
  - Simplifies clients, hence promotes the reduced coupling of features.
  - Improves scalability by virtue of the server off-loading work onto the clients.
- Trade off:
  - Reduces visibility generated by the code itself, which is hard for an intermediary to interpret.