Service-Oriented Computing (SOC)

Software as a Service

- A service is a piece of well-defined functionality that is
  - available at some network endpoint
  - accessible via various transport protocols and serialization formats
  - always on (no construction or destruction semantics)
- Focus is placed on the utility of services

SOC Software Requirements

- Technology neutral
  - use widely available, standardized technology
- Loosely coupled
  - minimize formal knowledge required to use a service
- Location transparent
  - be publicly discoverable
  - provide QoS regardless of location of service or clients

Web Services

- Platform independent, network accessible service
- Typically stateless (extensions exist)
- Name derived from traditional use
- Firewall friendly
- Machine-to-machine interaction (not human-to-machine)
- Focused on integration rather than performance
- Two main types: SOAP and REST
Web Service Data

- Platform independent
- Text resolved
- Short lived
- Limited in size (kilobytes)
- Descriptive

JavaScript Object Notation (JSON)

- A text-based open-standard data representation format
- A lightweight alternative to XML
- Designed for "human readable data exchange"
- Often argued to be more bandwidth-effective than XML

JSON Data Types

- Number (double)
- String (backslash-escaped unicode)
- Boolean
- Array (ordered sequence)
- Object (unordered collection of key-value pairs)
- null (empty)

JSON Schema Example

```json
{
  "phone": {
    "number": "555-4321",
    "required": true
  },
  "address": {
    "street": "E-street",
    "city": "L.A.",
    "zip": 12345,
    "required": true
  },
  "name": "John",
  "properties": {
    "name": "Contact",
    "items": {
      "type": "array",
      "description": "Properties of the contact",
      "required": true
    }
  }
}
```

Hypertext Transfer Protocol (HTTP)

- Text-based
- Application-level protocol (mostly used over TCP)
- Client-driven (requests and responses)
- Stateless (sessions stored in cookies / URL rewrites)
- Can handle text as well as binary data (encoded as text)
HTTP Request

- Request line (method + URI + protocol)
- Headers (request information)
- Body (optional)

HTTP Response

- Status line (protocol + status code + reason phrase)
- Headers (response information)
- Body (response data)

HTTP Request Methods

- HEAD - simulate a get request
- GET - retrieve resource
- POST - submit data to resource
- PUT - upload resource
- DELETE - delete resource
- TRACE - echo request
- OPTIONS - query server for supported methods
- CONNECT - create TCP/IP tunnel

HTTP Response Example

HTTP/1.1 200 OK
Date: Thu, 24 Jan 2011 12:31:00 GMT
Server: Apache/2.0.61 (Unix)
Accept-Ranges: bytes
Content-Length: 50
Content-Type: text/html; charset=ISO-8859-1

<html>
<head>
</head>
<body>
test
</body>
</html>

HTTP GET Request Example

GET /kurser/5DV095/VT11/test.html HTTP/1.1
Host: www.cs.umu.se

HTTP Transactions

1. A Connection is established
2. A HTTP request is received
3. The (logical) path in the request is translated
4. The requested resource is identified (via the path)
5. A server module handling that resource is invoked
6. The module processes the request and generates a reply
7. A mime-type is provided and a HTTP response is created
8. The HTTP response is sent (possibly in increments)
Java Servlets

- Java classes
- Implements the Java Servlet API interfaces (predates WS)
- Receives requests and generates responses
- Can be written manually
- Must be thread-safe
- Can be generated for many environments (e.g., JSP)
- Hosted in Servlet containers

REST Web Services

- REpresentational State Transfer (REST)
  - Originally described in PhD thesis by Roy Fielding (coauthor of HTTP v1.1)
  - Describes a web-like architectural style for modeling interaction with stateful resources using HTTP
  - Does not define representational models for resources
  - Outlines an architectural model, not standardized

RESTful Web Services

- Services seen as a collection of resources
- Defines
  - resource set base URI
  - resource representation MIME type (nouns)
  - resource operation set (verbs)
- Service functionality semantics implied by resource
  - Like SOAP, interface representation may differ from internal representation
- Also known as a RESTful web API

The Servlet service() method

- Part of a service pattern
  - init() - called on Servlet instantiation
  - service() - called for each request
  - destroy() - called on container shutdown
- Should not be implemented directly
- Inherit base class and implement handler methods
- Distinct handlers for each HTTP method (e.g., doGet())

1. service() parses request and determines HTTP method
2. service() calls appropriate handler method
3. Handler method processes request

Resource-Oriented Architecture (ROA)

- Classic client-server communication model
- Stateless protocols
- Focuses on resources
- Assumes standardized access protocols
- Allows caching of resource representations, and layers of communicating entities (client, proxies, server)

Design Principles

- Uniform identification of resources (URI)
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as engine of application state
- Freedom from choice in communication
- Freedom of choice in representation
**REST Web Services**

### CRUD Semantics

- Create - POST
- Retrieve - GET
- Update - PUT
- Delete - DELETE

<table>
<thead>
<tr>
<th>Operation</th>
<th>Collection URI</th>
<th>Member URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>List members</td>
<td>Retrieve representations</td>
</tr>
<tr>
<td>PUT</td>
<td>Replace collection</td>
<td>Update / Create resource</td>
</tr>
<tr>
<td>POST</td>
<td>Create entry</td>
<td>Add subordinate</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete collection</td>
<td>Delete resource</td>
</tr>
</tbody>
</table>

### Design

- REST is an architectural style
  - not an API or a standard
- Client-server model
- Stateless interface
  - no need to keep track of sessions
  - all required information attached in call
- REST data format?
  - HTTP with attachments
  - XML
  - JSON
  - plain text
  - binary
  - ... |

### Interface Design

1. Determine nouns (resource types)
   - verbs (POST, GET, ...) are already defined
2. Define semantics for POST, PUT, GET, DELETE
   - to enable caching, GET should be side-effect free
3. Choose a data binding
4. Define (and document) what error codes apply

### REST Advantages

- Simplicity
  - Stateless
  - Easy discovery through hyperlinks
  - HTTPS security
- Good tooling support
  - Only read HTTP (and typically XML)
  - Web browsers works as clients
- Good performance
  - Internet scalability
  - GET may utilize caching

### REST Disadvantages

- No formal (WSDL-like) interface
  - how to know what parameters to pass?
- No input/output data validation
  - cannot automatically generate clients
- Only request-response
  - hard to implement one-way/asynchronous messaging
- Poor support for advanced functionality
  - transactions, QoS, reliable messaging, fine-grained / end-to-end security, message routing
- No standardized failure models
- May enforce tight coupling between clients and servers
- Often misunderstood and misused

### Misconceptions

- REST is typecast because its practices are folklore. It's got no canonical documentation beyond a doctoral thesis which, like most holy texts, says little about how to apply its teachings to everyday life.
- Some people believe that REST is whatever, do you do, as long as SOAP is not used
- URI misuse: POST http://www.abc.com/r1?method=delete
- The only thing you can use an identifier for is to refer to an object. When you are not dereferencing you should not look at the contents of the URI string to gain other information - Tim Berners-Lee