SOAP, REST, WebAPI and Databases

1. maj 2018

Agenda

SOAP

- What is SOAP
- History
- Terminology overview
- Message Format
- Technical critique

REST

- What is REST
- History
- Architectural overview
- Application

WebAPI

- MVC vs. Web API

Databases

Simple Object Access Protocol

What is SOAP

SOAP provides the Messaging Protocol layer

It is an XML-based protocol

Built around three characteristics:

- Extensibility
  - Security and WS-routing are some extensions
- Neutrality
  - SOAP can operate over any protocol such as HTTP, SMTP, TCP or UDP
- Independence
  - SOAP allows for any programming model

What is SOAP (cont.)

The SOAP architecture consists of several layers of specifications for:

- Message format
- Message Exchange Patterns (MEP)
- Underlying transport protocol bindings
- Message processing models
- Protocol extensibility

SOAP is the succesor of XML-RPC (Remote Procedure Call)

History

1998 – SOAP was designed as an object-access protocol

1998 – XML-RPC launched

1999 – The specification was not made available until it was submitted to IETF

2000 – Version 1.1 of the specification was published as a W3C Note (Not standardized)

2003 – Version 1.2 became a W3C Recommendation

2003 – Acronym was dropped (Simple Object Access Protocol)

SOAP Terminology

SOAP specification can be defined to be consisting of 3 conceptual components:

- Protocol concepts
- Data Encapsulation concepts
- Message Sender and Receiver Concepts

These concepts defines SOAP

Protocol Concepts

SOAP:

- The set of rules formalizing and governing the format and processing rules

Nodes:

- These are physical/logical machines used to transmit/forward, receive and process SOAP messages.

Roles:

- All nodes assume a specific role. The role of the node defines the action that the node performs on the message it receives

Protocol binding :

- A SOAP message needs to work in conjunction with other protocols to be transferred over a network.

Features:

- SOAP provides a messaging framework only. However, it can be extended to add features such as reliability, security etc.

Module :

- A collection of specifications regarding the semantics of SOAP header to describe any new features being extended upon SOAP.

Data Encapsulation concepts

Message:

- Represents the information being exchanged between 2 soap nodes.

Envelope :

- It is the enclosing element of an XML message identifying it as a SOAP message.

Header block:

- A SOAP header can contain more than one of these blocks, each being a discrete computational block within the header. In general, the SOAP role information is used to target nodes on the path.

Header:

- A collection of one or more header blocks targeted at each SOAP receiver.

Body:

- Contains the body of the message intended for the SOAP receiver. The interpretation and processing of SOAP body is defined by header blocks.

Fault:

- In case a SOAP node fails to process a SOAP message, it adds the fault information to the SOAP fault element. This element is contained within the SOAP body as a child element.

Message Sender & Receiver Concepts

Sender:

- The node that transmits a SOAP message.

Receiver:

- The node receiving a SOAP message. (Could be an intermediary or the destination node.)

Message path:

- Path consisting of all the nodes that the message traversed to reach the destination node.

Initial SOAP sender:

- The node which originated the message to be transmitted. This is the root of the SOAP message path.

Intermediary:

- All the nodes in between the originator and the intended destination. It processes the header blocks targeted at it and acts to forward a message towards an ultimate SOAP receiver.

Ultimate SOAP receiver:

- The destination receiver of the SOAP message. This node is responsible for processing the message body and any header blocks targeted at it .

Message format

XML Information Set was chosen as the standard message format because of its widespread use by major corporations and open source development efforts

Typically, XML Information Set is serialized as XML

XML Information Set does not have to be serialized in XML. For instance, a CSV or JSON XML Information Set representation exists

The lengthy syntax of XML can be both a benefit and a drawback

XML promotes readability for humans, facilitates error detection, and avoids interoperability problems such as byte-order (endianness)

XML messages by their self-documenting nature usually have more 'overhead' (headers, footers, nested tags, delimiters) than actual

Message format example

```
<?xml version="1.0"?>

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:m="http://www.example.org/stock/Surya">

  <soap:Header>

  </soap:Header>

  <soap:Body>

    <m:GetStockPrice>

      <m:StockName>GOOGL</m:StockName>

    </m:GetStockPrice>

  </soap:Body>

</soap:Envelope>
```

Technical critique

Advantages

- SOAP's neutrality characteristic explicitly makes it suitable for use with any transport protocol. Implementations often use HTTP as a transport protocol, but obviously other popular transport protocols can be used.
- SOAP, when combined with HTTP post/response exchanges, tunnels easily through existing firewalls and proxies, and consequently doesn't require modifying the widespread computing and communication infrastructures that exist for processing HTTP post/response exchanges.
- SOAP has available to it all the facilities of XML, including easy internationalization and extensibility with XML Namespaces.

Disadvantages

- When using standard implementations and the default SOAP/HTTP binding, the XML infoset is serialized as XML. To improve performance for the special case of XML with embedded binary objects
- When relying on HTTP as a transport protocol and not using WS-Addressing, the roles of the interacting parties are fixed. Only the client can use the services of the other.
- The verbosity of the protocol, slow parsing speed of XML, and lack of a standardized interaction model led to the domination in the field by services using the HTTP protocol more directly. See, for example, REST.

Representational state transfer

What is REST

Web services are one way of providing interoperability between computer systems on the Internet.

REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.

Responses may be in XML, HTML, JSON or some other defined format.

Unlike SOAP-based Web services, there is no "official" standard for RESTful Web APIs

While SOAP is a protocol. REST is not a standard in itself, but RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML.

History

1996 – Development of REST initiated, based on HTTP 1.0

1999 – Development of REST ended

2000 – REST was defined by Roy Fielding

Architectural overview

Performance

- Component interactions can be the dominant factor in user-perceived performance and network efficiency

Scalability

- To support large numbers of components and interactions among components.

Simplicity

- A uniform Interface

Modifiability

- Components to meet changing needs (even while the application is running)

Visibility

- Communication between components by service agents

Portability

- Components by moving program code with the data

Reliability

- The resistance to failure at the system level in the presence of failures within components, connectors, or data[9]

Architectural constraints

There are six guiding constraints that define a RESTful system:

- Client-server
- Stateless
- Cacheable
- Layered system
- Code on demand (optional)
- Uniform interface

If a service violates any of the required constraints, it cannot be considered RESTful.

Constraint: Client-server

Must adhere to the client-server architectural style.

Separation of concerns is the principle behind the client-server constraints.

By separating the user interface concerns from the data storage concerns, we improve:

- portability of the user interface across multiple platforms
- scalability by simplifying the server components.

Perhaps most significant to the Web, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

Stateless

The client–server communication is constrained by 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.

The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication.

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.

Layered system

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

Code on demand (Optional)

Servers can temporarily extend or customize the functionality of a client by transferring executable code.

Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Uniform interface

The uniform interface constraint is fundamental to the design of any REST service.

It simplifies and decouples the architecture, which enables each part to evolve independently.

The four constraints for this uniform interface are:

- Identification of resources
  - Individual resources are identified in requests, for example using URIs in Web-based REST systems.
- Manipulation of resources through representations
  - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
- Self-descriptive messages
  - Each message includes enough information to describe how to process the message.
- Hypermedia as the engine of application state (HATEOAS)
  - Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available.

Application

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://api.example.com/resources/
- An internet media type that defines state transition data elements
- Standard HTTP methods (e.g., OPTIONS, GET, PUT, POST, and DELETE)

URL and HTTP Methods

URL	GET	PUT	POST	DELETE
Collection, such as http://it17.com/resources/	**List** the details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection.	**Delete** the entire collection.
Element, such as http://it17.com/resources/item1	**Retrieve** a representation of the addressed member	**Replace** or **Create** the addressed member	Not generally used.	**Delete** the addressed member

.NET Core Web API

MVC vs. Web API

Before ASP.NET Core, they were very similar. Both followed an MVC type pattern with controllers and actions.

Web API lacks a view engine (Razor) and instead was designed to be used for REST APIs.

MVC was designed for standard web applications with HTML front ends.

Microsoft touted Web API as a framework for building any type of HTTP service.

It was a great alternative to WCF, SOAP, and older ASMX style web services.

It was designed from the ground up with JSON and REST in mind.

.Net Core Web API

Old:

```
[Route("api/[controller]/[action]")]
public class ValuesApiController : ApiController
{
    [HttpGet]
    [ActionName("GetArray")]
    public IEnumerable GetArray()
    {
        return new string[]
                 { "value2", "value3" };
    }
}
```

New:

```
public class ValuesController : Controller
{
    public IEnumerable GetArray()
    {
        return new string[]
                 { "value2", "value3" };
    }
}
```

Databases

Hands-on Lab

Guide for Entity Framework:

https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db

Follow the step in the link above

Related Links:

https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro

In Memory db:

https://docs.microsoft.com/en-us/ef/core/providers/in-memory/

Entity Framework and MVC (Long guide)

https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro

Data access overview:

https://blogs.msdn.microsoft.com/dotnet/2016/11/09/net-core-data-access/

Introduction to Identity (authentication)

https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity

Questions?