COSC 331 – Microservices and Software

Fall 2020

Recap: APIs

- Last lecture, we discussed the who, what, where, when, and why of using an API
  - APIs provide a machine-readable interface for our service
  - They allow external developers and third parties to access our service in a programmatic way
- An API isn’t always necessary, but makes sense if we’re handling multiple functions or if you intend to make your service available to third parties

SOAP: The Original Middleware

- **SOAP**, or **Simple Object Access Protocol**, was originally developed in the late 90's, and eventually standardized by the W3C (The World Wide Web consortium) in the early 2000s (2003)
- SOAP became one of the first standardized inter-application protocols of the modern web
    - Widely implemented, and still frequently seen and used today, particularly in enterprise systems

Scratching the Surface

- It’s important to understand that SOAP is actually a broad standard that covers many different components:
  - A message format based on XML
  - A means of transporting messages via a variety of transport protocols like HTTP and SMTP
  - A set of rules used for the processing and transit of SOAP across multiple nodes
  - Definitions for converting an **RPC** (**Remote Procedure Call**) into a SOAP message and back again

The Structure of SOAP Messages

- SOAP is entirely based on the **eXtensible Markup Language (XML)** data exchange format
- A SOAP message contains three major parts:
    - An envelope – this is the “wrapper” of the message, which identifies it as a SOAP message
    - An optional header – used for passing along routing and processing information related to the message
    - A body – the actual payload of data that is to be delivered

Intermediaries in SOAP

- To give you an idea of how complex of a system SOAP can actually be, there are provisions for passing a SOAP message across multiple domains
- Between the sender and the receiver there may be additional processing “nodes” that may provide additional services and/or modify the SOAP message
  - The optional headers section of a SOAP message can contain complex routing and processing information for these intermediary services

The SOAP Body

- The SOAP message body is where the actual data “payload” is found
- This data will be one of two things:
  - Either the application-specific data that has been requested and is being passed along (i.e. what we would typically think of as the output of a service), **OR**
  - A fault message to inform the recipient that something went wrong
- This is either/or – faults and data cannot be sent together

SOAP via HTTP

- We’re going to ignore the other protocols like SMTP that SOAP can be used with, since they’re not particularly relevant here
- SOAP can use GET or POST requests, although in earlier versions of SOAP POST was the primary HTTP request type
- Depending on request type, two different situations:
  - GET request: non-SOAP request, SOAP response
  - POST request: SOAP request, SOAP response
- GET requests typically don’t have a body element, so no way to reliably send the SOAP-format XML → hence non-SOAP request

SOAP in Summary

- So what is SOAP in summary?
  - An envelope that describes the message format
  - An optional header that describes where the message is going and who needs to process it before it reaches it’s final destination
  - A payload body that contains either our application data, or an error message
- This is standardized and rigid – there a rules that must be abided by if you wish to use the SOAP messaging protocol

What SOAP is Good For

- SOAP is a product of it’s time period, before JSON really took off and when the World Wide Web was young
- But that isn’t to say SOAP is obsolete – it still makes sense in some contexts
- Large institutional data transfer, particularly between large-scale applications where everything needs to be well documented and standardized for enterprise reasons
  - Think banks, markets, and critical data applications

So Why Not Use SOAP?

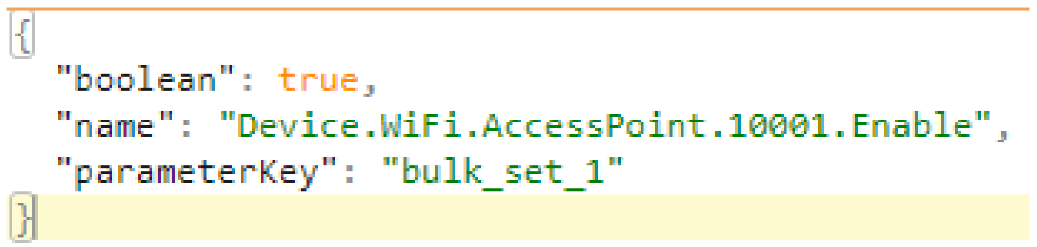
- In a microservice context, SOAP has some major disadvantages
- One of the biggest disadvantages from our perspective is the reliance on XML as the data exchange format
  - More of a pain to parse than JSON
  - Often requires fetching namespaces and vocabularies from remote sources before parsing can be done
  - A lot more overhead to send the same amount of data, which means larger messages

An Example of SOAP XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:cwmp="urn:dslforum-org:cwmp-1-0">
  <SOAP-ENV:Header>
    <cwmp:ID SOAP-ENV:mustUnderstand="1">112</cwmp:ID>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <cwmp:SetParameterValues>
      <ParameterList SOAP-ENC:arrayType="cwmp:ParameterValueStruct[1]">
        <ParameterValueStruct>
          <Name>Device.WiFi.AccessPoint.10001.Enable</Name>
          <Value xsi:type="xsd:boolean">1</Value>
        </ParameterValueStruct>
      </ParameterList>
      <ParameterKey>bulk_set_1</ParameterKey>
    </cwmp:SetParameterValues>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- Look at how much "stuff" we need, just to send two strings and a Boolean value!
- Schemas, encoding, and a lot of XML opening/closing tags that bulk up our message

A JSON Comparison



```
{
    "boolean": true,
    "name": "Device.WiFi.AccessPoint.10001.Enable",
    "parameterKey": "bulk_set_1"
}
```

- Here is all the same body data, formatted using JavaScript Object Notation (JSON)
- Granted: the JSON is missing some potentially important metadata that is part of the XML tags and schemas
- But do we need all that metadata? If not, then we're just wasting bandwidth when we use XML instead of a lighter-weight data exchange format like JSON

A RESTful Alternative

- So if not SOAP, then what? The answer is **REST**, or **Representational State Transfer** architecture
- Unlike SOAP, REST is not actually a “standard” at all, but a more loosely defined application architecture for communicating via HTTP
- As we discussed last lecture, REST has become the defacto standard for modern web APIs, and has displaced SOAP to a certain degree as the preferred way to communicate between applications

REST and the Web

- REST isn’t just a web technology, but is quite literally a product of the World Wide Web, and the HTTP communication protocol
- REST was specifically conceived as a way of leveraging existing HTTP protocols and the client-server architecture that is so common across the World Wide Web
- Originally proposed by Roy Fielding, who was also a major author of the HTTP specification (not a coincidence!)

The Three Components of REST

- Rest comprises of three major components, or concepts, that we need to understand:
  - **Nouns** – describe a resource, either plural or singular, that the client may access
  - **Verbs** – describe how we are interacting with a resource
  - **Representations** – describe how we are communicating information about a resource to the client

Nouns and Resources

- A **resource** in REST can be literally anything that could be described as information:
  - An image
  - Market data
  - An HTML page
  - Even non-digital objects, like equipment and people
- In REST, a resource is identified and represented by an identifier, a **Uniform Resource Identifier**

The Uniform Resource Identifier

- We see Uniform Resource Identifiers every day on the web. Here’s some examples:

Mfritter@okanagan.bc.ca

http://google.com

www.myapplication.com/vendors/memory/kingston

- Look at all those nouns! Each URI shares specific information to retrieve a specific resource

Verbs in REST

- Obviously if we have resources, we want to interact with them in some way
- Verbs, of course, describe **actions** – and in REST, those verbs are part of how we communicate, namely HTTP
- There are more request types in HTTP than just GET and POST:
  - PUT
  - DELETE
  - PATCH

HTTP Verbs and How They’re (Supposed) to Work

- The idea is that we use an HTTP request type (a verb) which is appropriate to what we’re trying to do
  - GET – fetches a resource
  - POST – create a new resource
  - PUT – replace an existing resource
  - DELETE – delete the specified resource
  - PATCH – partially modify an existing resource
- Combine any URI + any verb, and you should be able to do anything you need with the resources available

An Example

- Consider this URI:

  www.cats.com/orange/tabby/tabbyCat3.png
- This URI points to an image resource
- We could use GET to fetch the image, and render it for the user
- We could use DELETE to remove the image
- We could use PUT to upload a modified version of the image

Representations in REST

- The last major concept in REST is that of “representation” – how a resource is represented and returned to the client
- A resource can have multiple representations – consider an image in PNG format, rendered for the user, versus an image stored in a database, encoded in a textual format like base64
  - Same image, same resource, different representations
- Typically, for passing general data (ints, strings, etc), the representation for a REST API is typically JSON or XML
  - Of course, we’ll be focusing on JSON, for previously mentioned reasons

Constraint in REST

- Of these three components, we could say that two of them are "constrained", and one is unconstrained
- We are limited by the HTTP standards in what verbs we can potentially use – we don't want to go making our own HTTP request types
    - HTTP 1.1/1.2 actually included additional verbs specifically to help implement REST architectures
- Similarly, we're stuck with existing data exchange formats like JSON and XML
- Our nouns though (descriptors of resources) are unlimited – we can make our URIs as long as we want! (well, mostly)

Next Week

- Next week, we’ll continue digging into the REST architecture, and we’ll start looking at what exactly it means to call an API “RESTful”
- How to handle those strange HTTP request types like PUT and DELETE, which we rarely see in “normal” everyday web contexts
- Building our own RESTful APIs for microservices

Any Questions?