DSC 510

WEEK 9 INTRODUCTION TO WEB SERVICES

Web Services

Databases, web sites, and business applications need to exchange data. This is accomplished by defining standard data formats such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON), as well as transfer protocols or Web services such as the Simple Object Access Protocol (SOAP) or the more popular Representational State Transfer (REST). Developers often have to design their own Application Programming Interfaces (APIs) to make applications work while integrating specific business logic around operating systems, or servers. This section introduces these concepts with a focus on the RESTful APIs.

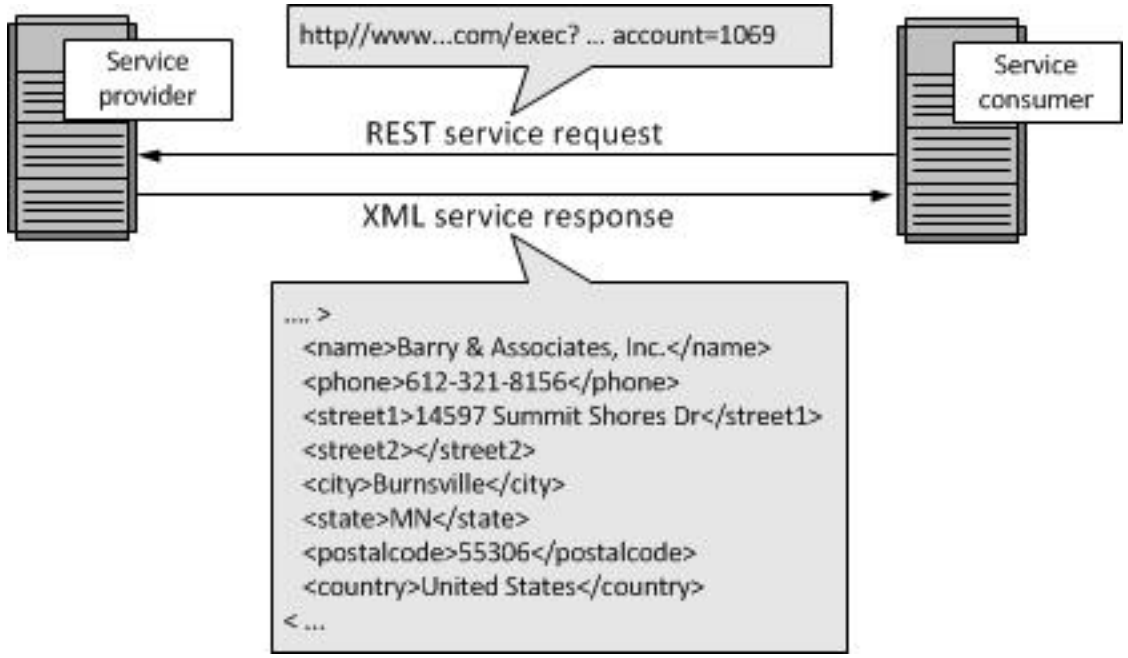
You might be asking yourself why would I care or why would I need to know this. The truth is that as a data science student you are going to get information in order to analyze it from somewhere. This data will not typically be sent to you in a CSV or text file. It will typically be sent to you in the form of a web service endpoint which you will access to obtain the data you need.

REST

The main idea behind REST is that a distributed system, organized RESTfully, will improve in the following areas:

- **Performance:** The communication style proposed by REST is meant to be efficient and simple, allowing a performance boost on systems that adopt it.
- **Scalability of component interaction:** Any distributed system should be able to handle this aspect well enough, and the simple interaction proposed by REST greatly allows for this.
- **Simplicity of interface:** A simple interface allows for simpler interactions between systems, which in turn can grant benefits like the ones previously mentioned.
- **Modifiability of components:** The distributed nature of the system, and the separation of concerns proposed by REST (more on this in a bit), allows for components to be modified independently of each other at a minimum cost and risk.
- **Portability:** REST is technology- and language-agnostic, meaning that it can be implemented and consumed by any type of technology (there are some constraints that I'll go over in a bit, but no specific technology is enforced).
- **Reliability:** The stateless constraint proposed by REST (more on this later) allows for the easier recovery of a system after failure.
- **Visibility:** Again, the stateless constraint proposed has the added full state of said request. From this list, some direct benefits can be extrapolated. A component-centric design allows you to make systems that are very fault-tolerant. Having the failure of one component not affect the entire stability of the system is a great benefit for any system. Interconnecting components is quite easy, minimizing the risks when adding new features or scaling up or down. A system designed with REST in mind will be accessible to a wider audience, thanks to its portability (as described earlier). With a generic interface, the system can be used by a wider range of developers. In order to achieve these properties and benefits, a set of constraints were added to REST to help define a uniform connector interface. REST is not suggested to use when you need to enforce a strict contract between client and server and when performing transactions that involve multiple calls.

REST Diagram



RESTful Web Services

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

REST gives us a standard way of allowing access to information via the web. The web server makes information (resources) available through its REST implementation. The client accesses these resources and using HTTP methods such as GET, PUT, DELETE, and POST. The web server provides information back to the client that the client can use. The data sent back to the client is generally JSON or XML data.

REST Principles

Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by a **Uniform Resource Identifier** (**URI**). A URI is a string of characters used to identify a resource. They provide a global addressing space for resource and service discovery. See The @Path Annotation and URI Path Templates for more information.

REST Principles

Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.

REST Principles

Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

REST Principles

Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and "Building URIs" in the JAX-RS Overview document for more information.

Web Service Use Case

- Modern programs use web services in a variety of ways. One example is with weather information. Imagine that you are writing an application which allows a user to display weather data for a specific location. You might write a program which obtains information from the user such as their zip code. Your program can then use a 3rd Party API (web service) in order to look up the weather for that zip code. Your program simply interacts with the user, requests information then uses the web service for the heavy lifting of the program. Your class project is to design a client that implements this example.
- In this class we use the Request library in order to make a HTTP request to a 3rd party web service.

API Example Code

```
import requests

url = "http://api.openweathermap.org/data/2.5/weather"

#This is the query string that we're going to submit to the URL.
#Notice that this is a dictionary with key value pairs.
querystring = {"zip":"68144",
               "APPID":"d5751b1a9e2e4b2b8c7983646072da8b"}
#This is used to submit any headers we need for the HTTP Request
#Notice that the headers are a dictionary.
headers = {'cache-control':'no-cache'}

#This is where the real magic occurs.  We're going to submit all of our data to the URL in order to get
#data back in the form of JSON data.  Keep in mind that the data you get back depends on the webservice
#that you interact with
response = requests.request("GET", url, headers=headers, params=querystring)
print(response.text)

```

API Response

This example shows the JSON response provided by the weather API. Notice how easy the data is to read. This is the power of JSON. Python makes processing JSON data so incredibly easy since it's really just a dictionary!


```
{
    "coord": {
        "lon": -96.12,
        "lat": 41.23
    },
    "weather": [
        {
            "id": 800,
            "main": "Clear",
            "description": "clear sky",
            "icon": "01d"
        }
    ],
    "base": "stations",
    "main": {
        "temp": 269.04,
        "pressure": 1029,
        "humidity": 92,
        "temp_min": 268.15,
        "temp_max": 270.15
    },
    "visibility": 16093,
    "wind": {
        "speed": 3.1,
        "deg": 330
    },
    "clouds": {
        "all": 1
    },
    "dt": 1520945700,
    "sys": {
        "type": 1,
        "id": 860,
        "message": 0.0044,
        "country": "US",
        "sunrise": 1520944709,
        "sunset": 1520987377
    },
    "id": 0,
    "name": "Omaha",
    "cod": 200
}
```

HTTP Methods AKA REST API Verbs

- When connecting to an endpoint you will typically use one of 5 methods depending on the task you're attempting to perform
  - GET – Used to retrieve information from the endpoint.
  - POST – Used to create a new entity on the endpoint or to update an existing entity.
  - PUT – Used to create a new entity or to update an existing entity. The most significant difference between POST and PUT is that PUT is idempotent.
  - DELETE –Used to request that a resource be removed.
  - PATCH – Used to request a partial update to a resource.
- **CRUD** stands for Create, Read, Update, and Delete. But put more simply, in regards to its use in RESTful **APIs**, **CRUD** is the standardized use of HTTP Action Verbs as defined above.

Fantastic explanation of Idempotence

From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

HTTP Status Codes

- When you make a request of a web service the web service will typically provide you with a status code. These status codes can be used to determine if a request was successful, if an error has occurred or other various operations.
- Common Status Codes:
  - 1xx – Informational
  - 2xx – Success
  - 3xx – Redirection
  - 4xx – Client Error
  - 5xx – Server Error
- When making a connection to an web service it is a good idea to validate the connection was successful with a try block before sending additional requests.

Creating Python Web Services

- It is also possible to create web services using Python. Creating web services are beyond the scope of this particular class however it's important to know that this capability exist and the power of this capability.
- It is certainly possible that you may at some point create a program which seeks to process data which is going to be consumed by another application. In this case you very well may wish to create a web service in order to allow other programs to obtain your data.

Popular API Frameworks for Python

- Django REST framework and Tastypie are the two most widely used API frameworks to use with Django. The edge currently goes to Django REST framework based on rough community sentiment. Django REST framework continues to knock out great releases after the 3.0 release mark when Tom Christie ran a successful Kickstarter campaign.
- Flask-RESTful is widely used for creating web APIs with Flask. It was originally open sourced and explained in a blog post by Twilio then moved into its own GitHub organization so engineers from outside the company could be core contributors.