COSC 331 – Microservices and Software

Fall 2020

Recap: Lab 2

- Lab 2 materials are available on Moodle/GitHub
- The Friday lab session was recorded w/ a complete walkthrough of the lab (last section of the lab was removed for debugging)
  - Moving forward I'll be continuing to record lab sessions, although I'll have to ask student permission (privacy concerns)
- Main takeaways – designing a multi-function microservice w/ Python, working with AJAX calls

What We've Seen So Far

- So far, we've looked at designing a simple microservice
  - Server loop, handling client requests, producing responses from our service
  - We've seen it in both Java and Python flavours now
  - Validating and processing user input
- Frontend design – client, server, microfrontend
  - Making asynchronous service calls with JavaScript
  - Producing HTML on demand from our microservice

What We're Missing

- We've looked at microservices and at frontends, but we haven't really discussed how we design the interface between those two components
- Enter the ***Application Programming Interface***, or ***API***
- An API is designed to handle the communication between two services, or a service and some other application
  - APIs are almost always served by services (always-on), but are often used by short-term applications (limited runtime)

What Makes an API?

- So what exactly makes an API, an API?
- An API typically has these main characteristics:
  - Communicates via a well-known protocol (like HTTP) and data exchange format (like JSON)
  - Is designed primarily for machine-readability, as opposed to human readability*
  - Serves as an abstraction – allows interaction, but hides away the implementation behind a “pretty” facade

Is an API always Necessary?

- ***NO!*** APIs can be useful in many circumstances, but they aren't free – they take time ($) to implement and maintain
- Ask yourself:
  - Does your service handle multiple functions or outputs, based on user/client input?
  - Will your service be used by people other than end users and internal developers (i.e. external devs, third parties)
  - Do you intend to provide your service as a product (or for free) to the general public and/or third parties?

Why APIs are Beneficial

- What an API offers is the ability for ***extension*** of your service by other developers and programmers
- For security (and organizational) reasons, we don't want to just let every developer out there into our source code and implementation
  - Tower of Babel scenario – each dev trying to write custom handlers for their specific project(s)
- Instead, we offer them an API – a standardized way of communicating with our service and using it's functionality, without touching the implementation aspect

The Return of Software Agnosticism

- I’ve hammered on about this for a while now, but one of the major reasons to use an API is to achieve ***software agnosticism***
- Specifically, as long as we use a common data exchange format and communication protocol, it doesn’t matter what language our implementation is in, and it doesn’t matter what language a third-party dev is working with: our service and their application can still easily communicate with each other

The Longevity and Maintenance Aspect

- Another important reason for APIs is the idea of decoupling the interface of a service from it's implementation
- Consider: You write a service in 2020 and promise it'll be available until at least 2040
  - By 2030 your implementation is hopelessly out of date
  - However, you can rewrite (or even completely re-do) the implementation of the service, while keeping the API the same
- End-users of the API are unaffected by changes to implementation, as long as the API ***does not change***

APIs and External Development

- You might wonder – why give external developers and third parties a nice interface to access your service? If it's a service you're primarily using for yourself, what is the point?
- If you're actually developing a commercial (or FOSS) service that you want to attract users, having an API can be a big selling point
  - Simple integration into an existing codebase without lots of refactoring and bespoke handler design = WIN
- In addition, your service is potentially having it's usability being extended for you, free of charge with no labour on your part

Designing an API

- So how do we go about designing an API? What should we focus on?
  - First, think: What functions/outputs is a user ***realistically*** going to want to be able to access in a vacuum?
    - Not all functionality needs to be part of the API, especially functionality that is mostly used internally
  - What functions/outputs would be useful to communicate with directly ***in a machine-readable way***?

A Very Important Concept – Machine Reading

- Why focus on ***machine readability*** in an API?
- An API is not (and never has been) designed to be consumed by humans
  - The Graphical User Interface (GUI) is pretty much standard for human-oriented interfaces these days, and it's basically ***the opposite*** of an API
- If you're designing an API to be consumed by humans, you've already made a mistake – make a readable HTML page instead. Your users will thank you.

Readability and APIs

- Of course, that said, we shouldn't design APIs to be antagonistic towards humans
- APIs should produce machine-readable output, but should still be comprehendible by humans
- Consider this API call:

```
mySite.com/numbers/fetch/10/20/desc
```

- What is this API call supposed to be requesting? Who knows? This URI is opaque, the underlying function of these words and numbers isn't immediately clear to us

Self-Documenting APIs

- What if instead, our API call looked like this?

```
mySite.com/getNumbers?greaterThan=10&lessThan=
                                20&orderBy=desc
```

- You might think this looks messier with a query string attached, but *this* version provides far more information about what this call is actually ***doing***
- APIs should always be documented (like any code), but a good API doesn't require the user to constantly reference it when performing basic tasks

Removing Redundancy

- While it might *look* impressive to list a whole bunch of URIs and functions on your API documentation, is it really necessary?
- We should eliminate redundant functions from our API – even if they exist in our implementation
  - Consider: You have one function to return a generic object X, and another to generate a specific variant of object X that is used in a particular application or task
  - If the end user can use the generic function to build the specific version, there is no reason to include both in our API – include only the ***more general of the two***

Good Practice in API Management

- If you're providing an API, you have to take into account that you are now being relied upon by others to provide a service
- We, as users, really don't like it when services change unexpectedly, and we especially hate it as developers when those changes break our code
- You should consider API dev to be one-way only: you can always make your API larger, ***but you should avoid, if possible, removing or changing existing API interfaces***
- Remember: the implementation can change, but the interface the user/third party dev sees shouldn't

API Design Paradigms

- Luckily, people have thought about all these issues (and many more), and have developed design paradigms over the years to help shape APIs
- Two major contenders that we'll be taking a look at: ***REST*** and ***SOAP*** – REST is an architecture style, while SOAP is an actual set of defined protocols
- These paradigms are designed to help "standardize" APIs – not necessarily in a truly interchangeable way (since function names and the like are not standardized), but such that we can assume certain characteristics of the API

SOAP – the OG API Paradigm

- **SOAP** was among the first major API design paradigms to emerge in the late 90's – **Simple Object Access Protocol**
- More tightly coupled to certain technologies due to the time period – designed entirely with XML (eXtensible Markup Language) in mind, as JSON hadn't really taken off yet
- Designed to operate on any network protocol, even ones not traditionally associated with APIs (such as SMTP – Simple Mail Transfer Protocol)
- Very verbose – lots of overhead due to data markup, idea of 'enveloping' the data to increase cross-compatibility

REST – the Modern Wunderkind

- SOAP laid the foundation, but it's roots in XML and the 90's meant that it was ripe to get replaced when Web 2.0 came along
- Enter **REST – Representational State Transfer**
- You've probably heard of it before, for a time it was very houte-couture to say your application used REST interfaces (now it's just mundane – sigh)
- Focuses on HTTP communication using self-describing verbs to handle different types of request and response handling

Next Lecture: REST & SOAP

- In the next lecture, we'll start to actually dig into REST and SOAP
  - How they differ, and why REST is the predominant way of designing an API today
  - Shortcomings of REST – all those verbs that we don't hear about much – PUT, PATCH, DELETE?
    - Why a lot of so-called "RESTful" interfaces are actually more like REST-lite
  - How we can design our own RESTful APIs easily

Any Questions?