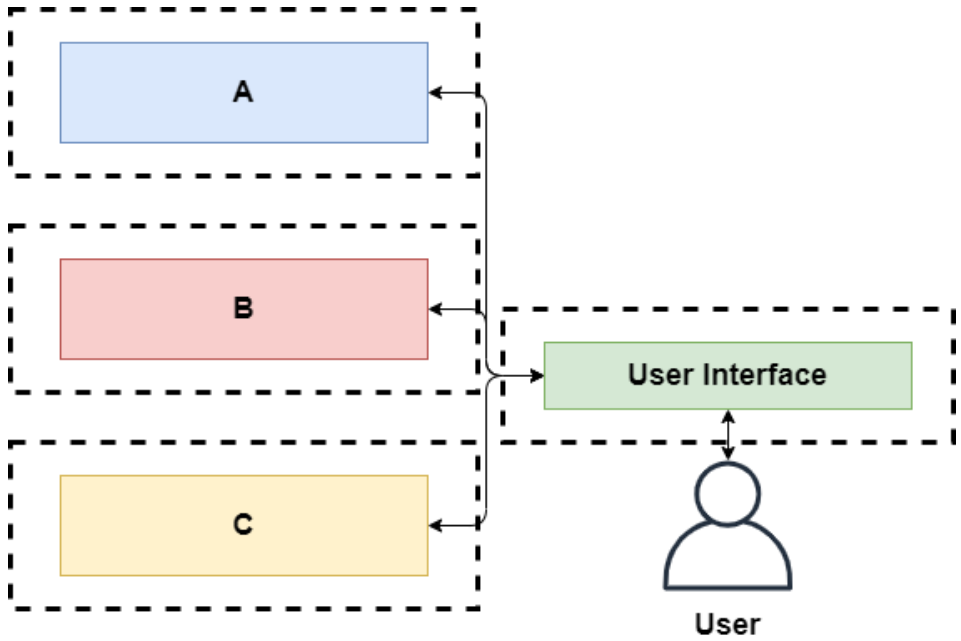
COSC 331 – Microservices and Software

Fall 2020

Announcements & Housekeeping

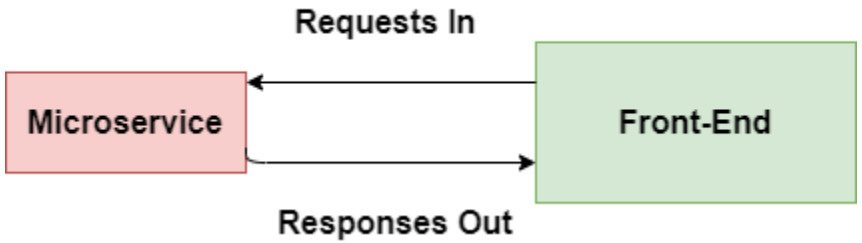
- ***We'll have our first quiz this week*** – online (Moodle), will be available on Wednesday and open for a period of one week
  - Will cover the notes up to (and including) Wednesday – microservice architecture and front end design
- Lab 1 – due on Thursday (11:59PM)
- Lecture 3 – notes posted and lecture recording available on Moodle

Recap: Microservice Architecture



- Last week, we talked about **microservice** architecture and how to go about designing a microservice
- We also looked at how to implement our first basic microservice – the CSS customizer

Front-End Design



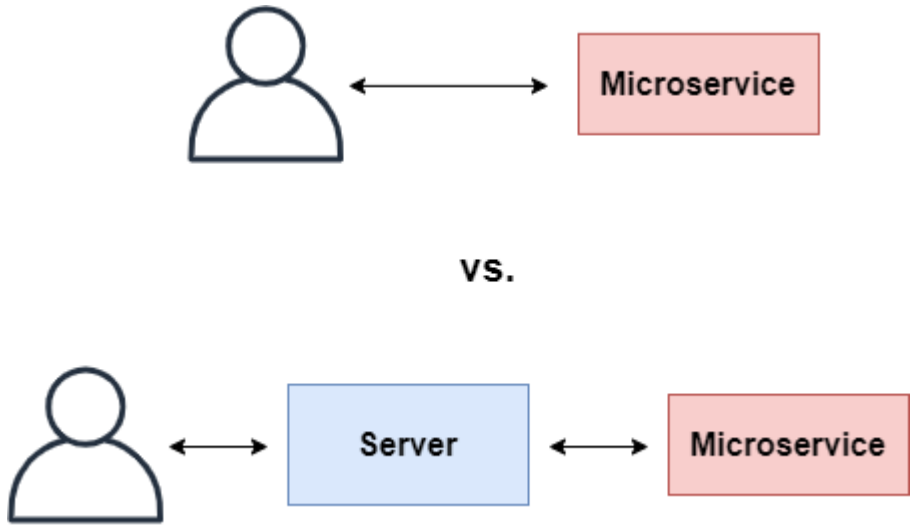
- In order to make our microservices useful, we'll need some sort of ***front-end*** to communicate with them
- Think of microservices as transmitters, and the front-end as a receiver
  - We can 'select a channel' – pick a microservice to communicate with and send out the appropriate request, then listen for a response

Different Types of Front-end Design

- The design of the front-end will vary depending on the specific needs of the project
- Things to consider when designing the front-end for your microservices:
  - ***Client-side*** vs ***Server-side***
  - Complete object vs. just data
  - Platform/Software agnosticism
  - Security

Client vs. Server Side Front End

- One of the first things you should consider is whether you intend for clients to directly interface with your microservices
- Will the client communicate directly with the microservice, or will there be another network component in between?
- Implications for security and microservice design



Client-side Front-end Design

- A client-side front-end means that the client is directly communicating with our microservice by way of some script or application ***stored on the client's machine***
- This could be JavaScript run inside the user's browser, or it could be part of a standalone desktop application
  - A good example is software update-checking services à la Notepad++ or Steam
- Client handles making requests and processing responses

An Example of a Client-side Front-end

- We've actually already seen a basic client-side front-end in the demo.html file in Lab 1

```
function fetchCSS(){
        var main = document.getElementById("main").value.substring(1);
        var accent = document.getElementById("accent").value.substring(1);
        var font = document.getElementById("font").value;
        var container = document.getElementById("container");
        var styleSheet = document.getElementsByTagName("link");
        container.style.display='none';
        styleSheet[0].href = "http://127.0.0.1/?main="+main+"&accent="+accent+"&font="+font;
        setTimeout(function(){
                container.style.display='initial';
        }, 2000);
}
```

- Simple JavaScript function takes user input from HTML5 form, and uses it to generate a request by way of making an href reference with a GET query string

The Benefits of Client-Side Design

- Client-side has some advantages:
  - Fewer total components – no need for a middleman server or application
  - More responsive – requests from the user immediately go to the microservice, and responses are returned directly to the client – no delay while waiting for the middleman
  - Allows load to be taken off of application servers, and transferred to microservices instead

The Dark Side of Client-side Design

- Remember the golden rule of dealing with user input:

Never Trust User Input!

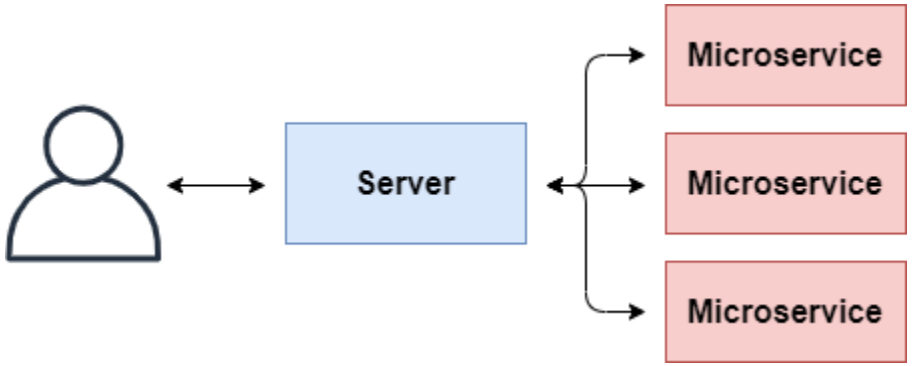
- Allowing the client to directly communicate with the microservice means we need very robust input validation, as we can't trust users to not try to break our service
- Furthermore, the client-side space is potentially ***hostile*** – we cannot trust anything stored there, as it can potentially be modified by the client – application state should not rely on client-stored data

Where Client-Side Makes Sense

- A client-side microservice front-end makes sense if we're serving client-specific, ***non-script*** files or data that doesn't require much (if any) additional processing
- Lab 1 is a good example of this – our basic microservice handles Cascading Style Sheets (CSS), which are client-specific and not a script file (although CSS does have some security exploits)
  - Consider: if a malicious user chooses to abuse the service, the only person they're hurting is themselves, as the CSS is only returned to (and rendered by) the user themselves

The Server-side Alternative

- When we're handling data that isn't appropriate to directly send to the client, or if we're handling multiple interdependent microservices, a server-sided front-end architecture may make more sense
- User communicates with a middleman server/application, which in turn communicates with the microservices



Benefits to Using a Server-side Approach

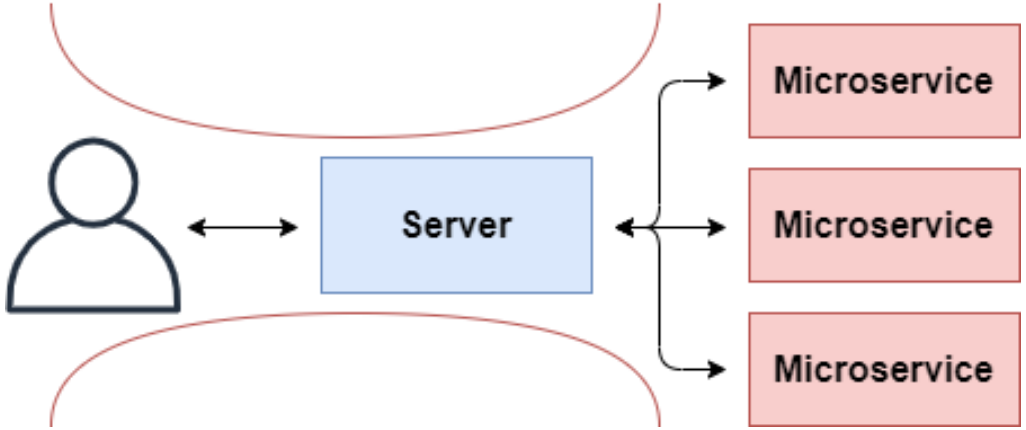
- Using a server-sided approach comes with some benefits:
  - It provides a unified service to the user – the server can communicate with multiple different microservices and present a single, unified interface to the client
  - Input can be validated when it arrives at the front-end, reducing the validation requirements of our microservices
    - Importantly, it also means we don’t necessarily need to make our microservices public-facing – private networking between microservices and front-end

Where Server-Sided is Necessary

- If we're dealing with potentially sensitive information (i.e. login information), using a server-sided front-end is usually a necessity
  - We don't want to trust the user to maintain application state information (whether a user is logged in, for example)
- Where microservices are **interdependent**, it may also make sense to use a server-sided approach to enforce ordering and data passing between different microservices

The Major Drawback to Server-sided

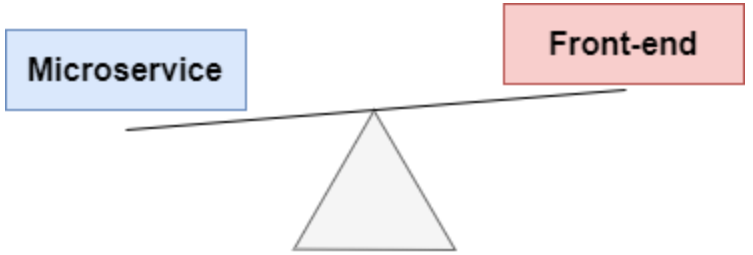
- The entire point of using a microservices architecture is to produce a scalable, loosely-coupled application
- If the server-side front-end isn't also loosely-coupled and scalable, we risk losing the advantages of a microservice by introducing a singular failure point and bottleneck for traffic



How to Prevent Bottlenecks

- Our server-side front-end should be scalable – we should design it in such a way that we can duplicate it and have multiple instances (servers/applications) running at the same time
- In addition, it should be ***fault-tolerant*** – if a microservice fails to provide a response in time due to being overwhelmed or down, the front-end should continue to function
  - Maintaining isolation between microservices

Balancing The Load



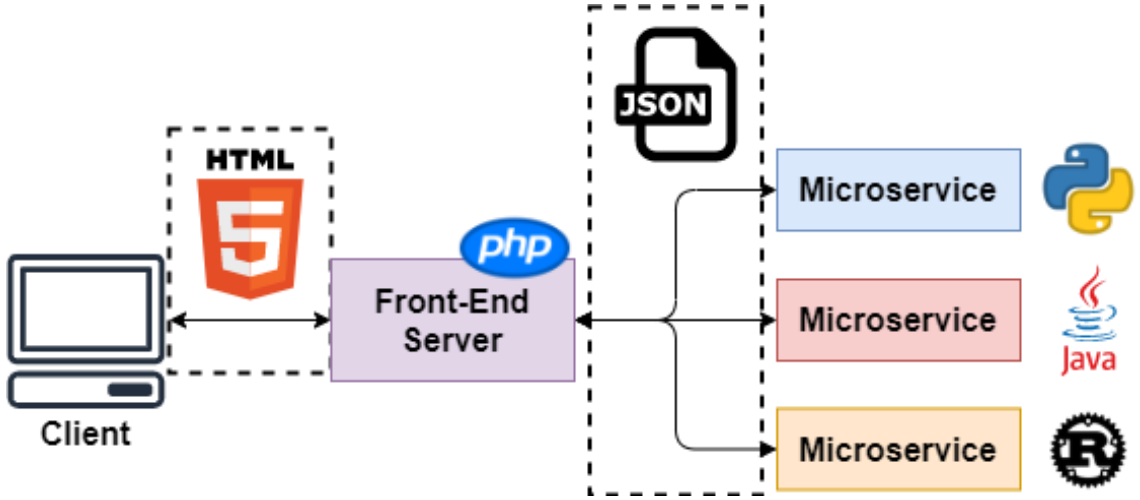
- How should we balance the load between the microservices and the front-end?
- In general, it makes more sense for most of the load to be carried by the microservices, as opposed to the front-end
  - It’s more efficient to scale the individual microservices that require more throughput than to scale the front-end
- Our front-end should be lightweight and perform minimal processing of the microservice outputs – processing should be moved to the microservices where possible

The Magic of Software Agnosticism

- Recall that one of the defining criteria of a microservice (first lecture) was that it typically communicates via an **API** of some sort, and generally uses a **common protocol** for communication, like HTTP
- Why is this important? Because it allows easy interfacing of different software components through the use of a shared format for data
  - Specifically, the use of a data exchange format (like JSON or XML) and protocol (HTTP) which can be used and understood across a wide spectrum of languages and software

A Diverse Microservice Ecosystem

- We can use common data exchange formats and communication protocols to build very diverse microservice ecosystems
- Almost all languages will have some in-built support for handling HTTP requests/responses and parsing JSON



Preview – This Week's Lab

- This week, we'll be building a less bare-bones microservice using Python
  - We won't need to work with sockets and do our own request handling – we'll use a proper threaded HTTP module instead
- Then, we'll be designing a simple client-side front-end using JavaScript
  - This front-end will combine our Java microservice from Lab 1, as well as our new Python microservice, allowing the client to communicate with two different services running two different languages at the same time

Any Questions?