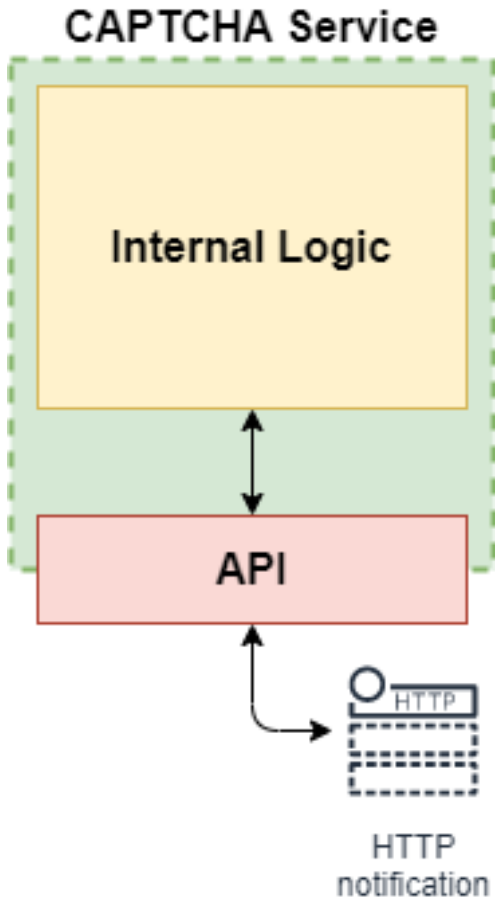
COSC 331 – Microservices and Software

Fall 2020

Recap: Designing Microservices

- Last lecture, we talked about the major components of a microservice
- We looked at a fairly abstracted model, constructed with an API component and an internal logic component
- The API handles input and output of the service, while the internal logic does the actual processing



Microservices in the Real World

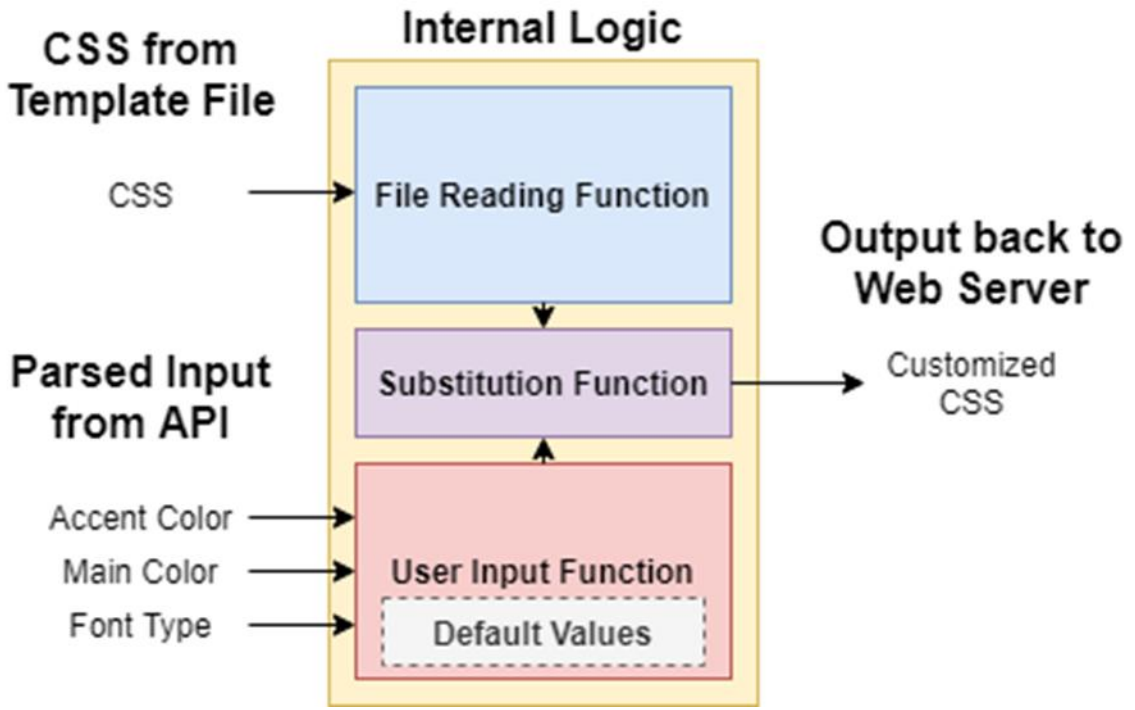
- Of course, this is a very simplified idea of a microservice – it doesn't include the sub-components of the API and internal logic
- This is a useful way to model and plan a microservice, but it doesn't tell us anything about the actual implementation
- In reality, the lines between components are often blurred – is the web server part of the API? Where is the line drawn between input parsing (API) and the internal logic?

Building an Actual Microservice

- As previously discussed, we're building a very basic microservice for Lab 1 using Java
- We'll use this as an example of what an actual microservice implementation might look like
- To make things simpler, we'll make our code modular, and start with the internal logic processing first
  - Why might it be a good idea to start with the internal components, before handling the API?

Processing our Inputs

- For this particular lab, our backend code is going to handle two different types of input:
  - A CSS file which serves as our base template
  - A set of between 0 and 3 values parsed from user input



Handling The CSS File

```
1   html {
2       font-family: _FONT_FAMILY_;
3   }
4   body {
5       background-color: _MAIN_COLOUR_;
6   }
7   .messageDiv {
8       background-color: #F5F5F5;
9       border: 3px solid _ACCENT_COLOUR_;
10      border-radius: 20px;
11      padding: 25px;
12      box-shadow: 5px 5px 5px 1px rgba(100, 100,
13  }
```

```
private static String readCSS() {
    try {
        BufferedReader br = new BufferedReader(new FileReader("src/template.css"));
        String line;
        String output = "";
        while ((line = br.readLine()) != null)
        {
          output = output + line;
        }
            br.close();
        return output;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

- The CSS file will serve as our base template, which our user input will be substituted into
- Nothing too fancy here: a basic buffered reader which reads the entire file into a string and returns it

Thinking About Optimization

- When designing a microservice, effort should be made to make the service as efficient as possible
- This includes both processing speed, as well as the output data size
- If we can avoid sending unnecessary or redundant data, we can shrink our output size, which in turn means shorter loading times and a reduced network load
- In what way could we make our CSS-handling service more efficient?

Minifying our Output

```
html {font-family: _FONT_FAMILY_;}body
{background-color: _MAIN_COLOUR_;}.messageDiv
{background-color: #F5F5F5;border: 3px solid
_ACCENT_COLOUR_;border-radius: 20px;padding:
25px;box-shadow: 5px 5px 5px 1px rgba(100, 100,
100, 0.2);}
```

```
output = output + line.replaceAll("\t", "");
```

- This CSS file is the same as the previous example, but only takes 234 characters, as opposed to 265 characters – a >10% reduction!
- We can remove tabs and newlines and still have a functional CSS file, just shorter and less readable
- This is known as ***minification***, and it's a common technique for reducing the overhead of CSS and JS files

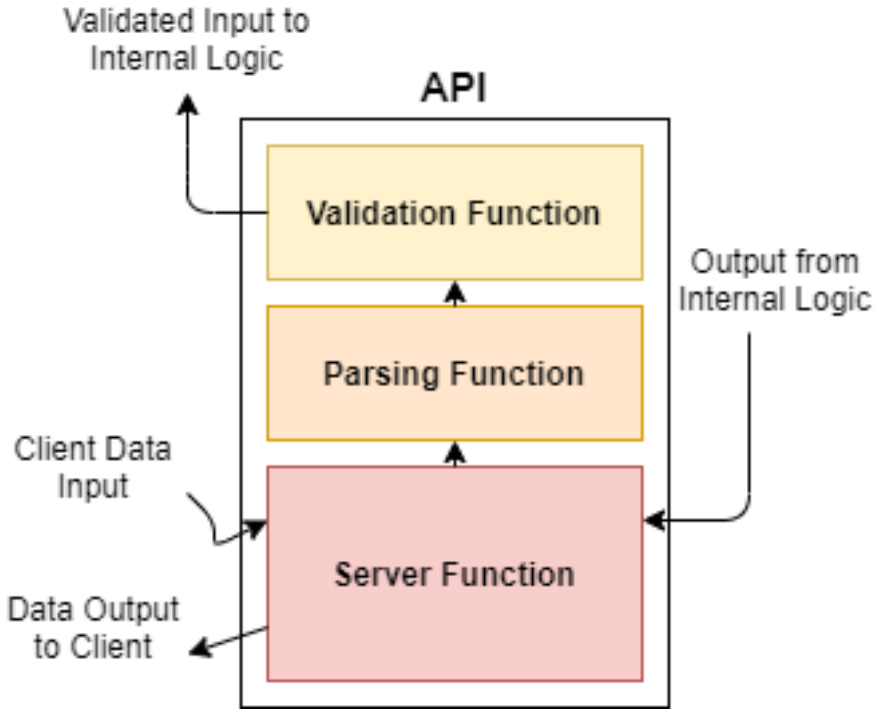
The Substitution Function

- The substitution function is very straightforward – we pass it our template CSS, along with our three user-inputted strings, and we replace the placeholder values in the template with them
- Don’t forget to include the hash symbol in front of the colours, to denote that we are using hex colors

```
private static String substituteCSS(String css, String accentC, String mainC, String font) {
    css = css.replaceAll("_MAIN_COLOUR_", "#" + mainC);
    css = css.replaceAll("_ACCENT_COLOUR_", "#" + accentC);
    css = css.replaceAll("_FONT_FAMILY_", font);
    return css;
}
```

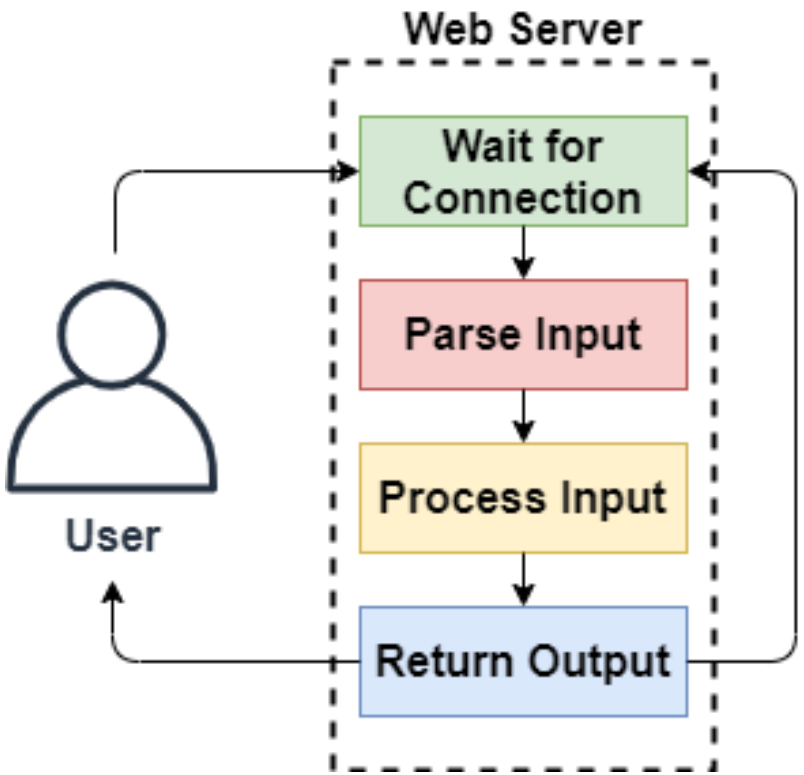
Developing the API

- The backend internals are basically complete, now we need to turn our attention to the API
- The API component of our microservice will handle three major requirements:



1. Handling incoming and outgoing connections and transmission
2. Parse incoming data to make it intelligible
3. Validate incoming data to verify it is acceptable

Managing Connections



- In order to handle the HTTP connections our microservice will use to communicate, we'll need a web server
- The web server simply waits and listens for connections, then dumps out the input to be further processed
- Once processed, the data is passed back to the server and sent back to the client using the original connection

A Simple Socket-based Server

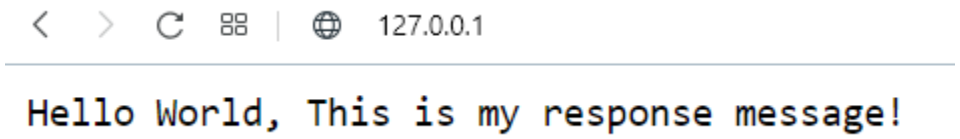
```
public static void webServe(int port) throws IOException {
    ServerSocket server = new ServerSocket(port);
    while(true) {
        Socket userConn = server.accept();
        BufferedReader br = new BufferedReader(new InputStreamReader(userConn.getInputStream()), 1);
        String output = "";
        String line;
        while ((line = br.readLine()) != null) {
            output = output + line;
            if (line.isEmpty()) {
                break;
            }
        }
        System.out.println(output);
        userConn.close();
        br.close();
    }
}
```

- Here is an example of a simple server using Java Sockets
- It listens on a specified port for incoming connections, then provides a buffered reader to handle incoming data, which it prints to the console
- Problem: No response is provided to the client!

Responding to Requests

- We can use the socket output stream to send a response back to the client using the original connection
- We have to send an HTTP status code (200 OK), and we use \r\n\r\n (also known as CR LF) to end the status code
- Add in our content to return afterwards, and then write it as bytes to the output stream

```
String helloWorld = "Hello World, This is my response message!\r\n";
String response = "HTTP/1.1 200 OK\r\n\r\n" + helloWorld;
userConn.getOutputStream().write(response.getBytes("UTF-8"));
userConn.getOutputStream().flush();
userConn.close();
```



Parsing Incoming Data

```
String pattern = "GET /.*" + variable + "=(.*?)(?:&| HTTP)";
Pattern regex = Pattern.compile(pattern);
Matcher match = regex.matcher(rawInput);
if(match.find()) {
    return match.group(1);
} else {
    return null;
}
```

- We can fetch the query parameters containing our color and font data using a fairly simple regular expression
- Our parse function will take a String variable, containing the name of our parameter (i.e. “accent”), and a String containing our raw client input
- The function will return null if the parameter is not found

Handling Our User Input; Or: Idiot Proofing

- There is a very important cardinal rule in working with user input: ***never trust user input***
- Consider: We expect a String containing a legitimate hex color, something like F5F5F5 for example
  - But what if, by accident or on purpose, our hex color input isn't a hex value at all, but something like ZALGO TEXT ?
- Before we do anything with our user input, we need some way of validating it
  - This is important for both security and stability reasons

Verifying The Color Values

```
private static String verifyHex(String rawInput, String defaultValue) {
    try {
        Long.parseLong(rawInput,16);
        if(rawInput.length() <= 6) {
            return rawInput;
        } else {
            return defaultValue;
        }
    } catch (Exception e) {
        e.printStackTrace();
        return defaultValue;
    }
}
```

- We expect the color values to be passed as a 6-digit hexadecimal number, although it **could** be less than 6 digits
- We can use parseLong with a base of 16 to check if our data is hexadecimal
- The validator should also check the length to make sure it’s less than or equal to 6 characters long

Verifying The Font Value

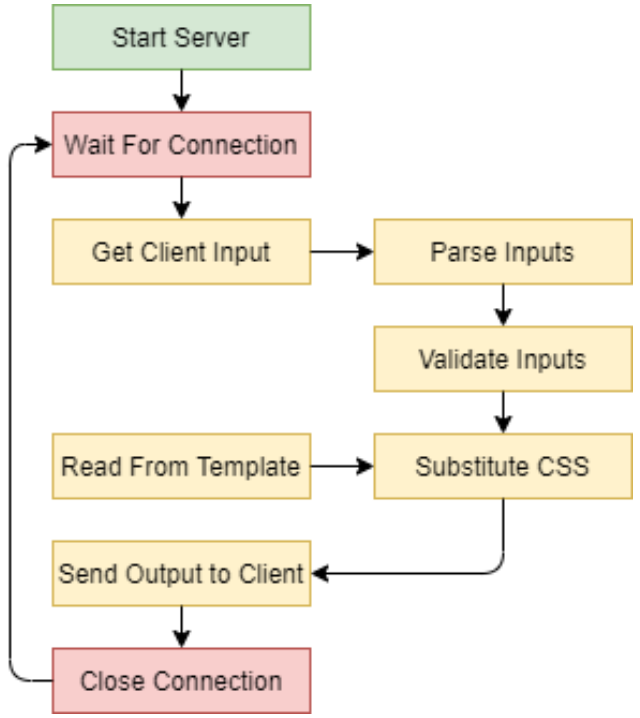
- The font value is simpler to validate: we're only considering the three generic font families available in CSS: monospace, serif, and sans-serif
- We simply have to check if the user input matches one of these three values; if so, we use that value, otherwise we use a default value
- The font validator function is left as an exercise to implement; you should use a similar function signature to the verifyHex function for simplicity

The Validation First Strategy

- In general, we should try to validate our data as soon as it is received from the client, and before it goes anywhere near our internal/backend code
- It is generally best to try to keep ***unsanitized*** (i.e. not yet validated) user input well away from your internal logic, to prevent unexpected errors from incorrect input, or to prevent malicious abuse
- In general, it's easiest to validate early, as you can then assume your backend code will only have to deal with an 'approved' subset of inputs (don't forget to catch null!)

Putting It All Together

- Now all of the components have been prepared, we can put together our microservice
- Everything needs to go inside the web server loop – we start by getting input, and should end by sending output and then closing our connection
- I recommend you build and test each function individually first, as it'll make debugging easier

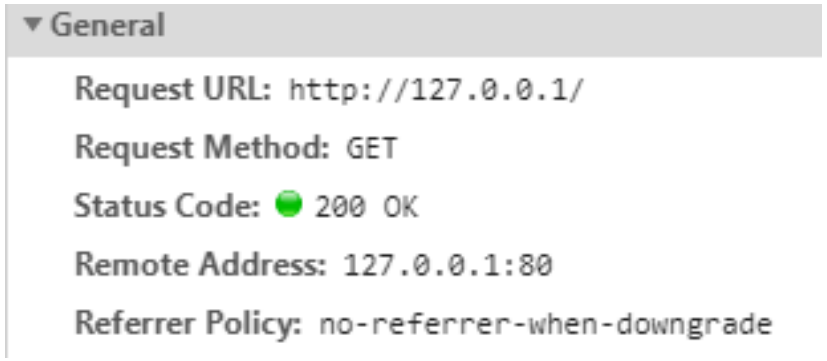


Testing it Out

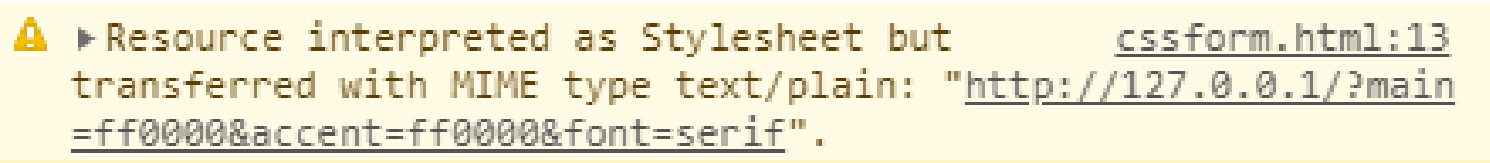
- With your server running, try visiting 127.0.0.1 in a web browser – you should see something like this:

```
html {font-family: sans-serif;}body {background-color:
#AAAAAA;}.messageDiv {background-color: #F5F5F5;border: 3px
solid #AAAAAA;border-radius: 20px;padding: 25px;box-shadow: 5px
5px 5px 1px rgba(100, 100, 100, 0.2);}
```

- A quick look at the Chrome inspector shows that we're communicating with our server:



Changing the Content Type



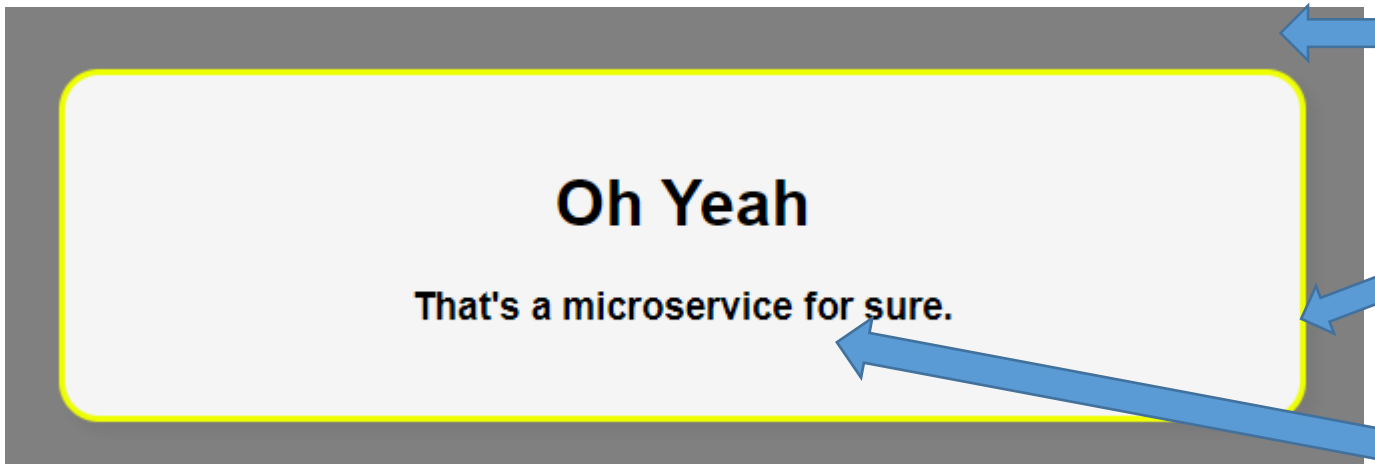
- While testing, you may notice a warning like this, stating that our data is being received as plain text instead of as CSS
- Most browsers can handle this, but it's better to remove the ambiguity – we just need to set the appropriate Content Type header to "text/css":

```
String response = "HTTP/1.1 200 OK\r\nContent-Type: text/css\r\n\r\n" + helloWorld;
```

See It In Action

- You can use the demo.html file to visualize the CSS changes in actual use
- It uses JavaScript to pass along your style choices as GET parameters in the CSS linkage, then redraws the page with the new styling*
    - *It'll flash a blank screen while redrawing – this is normal
- If the microservice is working correctly, you should be able to see the background and border accent colors change, as well as the font type

Demo.html Example



- Main Color
- Accent Color
- Using the “Sans-Serif” generic font family

Shortcomings of our Service

- As you can see, it’s not that difficult to build a simple microservice – this CSS customizer can easily be written in <100 lines of code
- However, this basic microservice has some major shortcomings due to it’s simplicity
  - Unthreaded operation – only one client can be served at a time
  - Extremely basic HTTP functionality – doesn’t handle unexpected requests well (always sends out CSS file regardless of actual request content)

Moving Forward

- Despite these shortcomings, this basic example is good for demonstrating the moving parts that make up a microservice – things like parsing, validation, and handling HTTP data
- For the next microservice we build, we'll use a pre-built web implementation with threading to make sure that our service can handle real-world user load
- So far, we've been running in a local environment – we'll start to move to online hosting for our services, and from there we can begin talking about containerization

Any Questions?