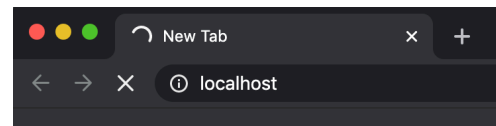
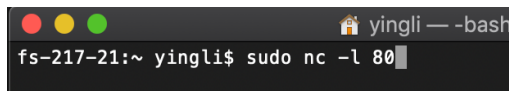


Application Layer (IV)

More GET Examples

- Let's take a look at the details of a http request. In this example, we use localhost as a server and send a http request to itself. You may get different information in the request when using different web browsers.



```

fs-217-21:~ yingli$ sudo nc -l 80
GET / HTTP/1.1
Host: localhost
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Google Chrome";v="89", "Chromium";v="89", ";Not A Brand";v="99"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.82 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
  
```

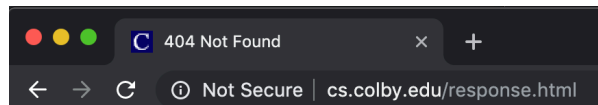
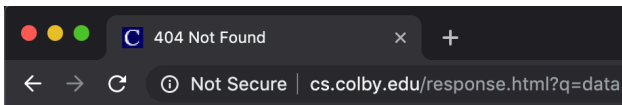
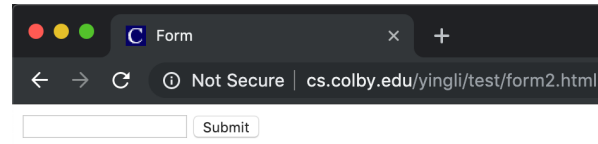
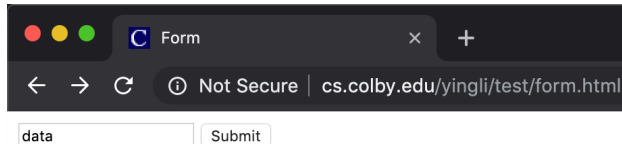
- Let's understand how both sides exchange information over HTTP using Wireshark.
 - Open Wireshark and monitor the active interface.
 - Open CS331 project 2 page using a web browser.
 - filter ip.addr == your laptop ip
 - GET request is the packet with GET /courses/S21/CS331/projects/p2.html HTTP/1.1
 - GET response is the packet with HTTP/1.1 200 OK
 - There is a GET for CSS file after the response without reopening the connection, which shows that this is a persistent connections.

HTTP POST Method

- POST
 - client informs server about data to be delivered
 - client sends data
 - e.g., submit an online form

Compare POST with GET

- request form1.html and form2.html on CS325 web server from a web browser and show the difference on the URL.
 - form1: input data is in the URL
 - form2: input data is not in the URL



Not Found

The requested URL /response.html was not found on this server.

Not Found

The requested URL /response.html was not found on this server.

- show form1.html and form2.html

```

form.html
< > form.html > No Selection
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Form</title>
5 </head>
6 <body>
7   <form action="/response.html" method="GET">
8     <input type="text" name="q">
9     <input type="submit">
10  </form>
11 </body>
12 </html>

```

```

form2.html > No Selection
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Form</title>
5 </head>
6 <body>
7   <form action="/response.html" method="POST">
8     <input type="text" name="q">
9     <input type="submit">
10 </body>
11 </html>

```

- form.html uses GET, so the submitted data was in the URL.
 - form2.html uses POST, so the submitted data was sent to the server implicitly (not in the URL).
- But both requests couldn't get appropriate responses, as the CS325 web server was not set up to response these requests.
 - So how can we set up a server?

Server Architecture

- Generally speaking, the server accepts and process the incoming requests while listening at a port.
- Typically, there are two ways to process the accepted requests, sequentially and in parallel.
- The web servers in real life must accept concurrent requests. So, processing the requests in parallel is a more feasible way.
 - The server can allocate a process or thread to handle a request, the main process and thread of the server deals with accepting the incoming requests.
- Event-driven is another approach to handle concurrent requests.
 - The server does a **quick bit of processing** and **schedule a callback when** (typically) **I/O is done**.
 - Node.js (<https://nodejs.org/en/>) provides a framework to implement an event-driven server.
 - **Not for CPU intensive application**.

Node.js Example 1

```
var http = require('http');

http.createServer(
  function (req, resp) {
    resp.writeHead(200, {'Content-Type': 'text/plain'});
    resp.end('Hello World\n');
  }
).listen(8000);

console.log('Server running at http://localhost:8000/');
```

- Show the server1.js code, run it, and request localhost:8000 on web browser.
- server1.js is a very simple server that displays a string and nothing more than that.
- To use the HTTP server and client one **must require('http')**
- **http.createServer(requestHandler)** returns a new instance of `http.Server`.
 - The requestHandler is a function that is called once a HTTP request is made against that server. Node calls the handler with two objects, request and response.
 - The request object has a url attribute that contains the pathname and the query.
 - The response object includes the HTTP status code, the response header, and the response content (e.g., the requested file).
- **writeHead(status code, header)** send a response header to the request. **Must be called before resp.end()**.
- **resp.end(data)** signals to the server that all of the response headers and body have been sent. **Must be called on each response**.
- Reference: <https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/>

Node.js Example 2

- The server should listen to the client request and act according if we want some interaction like sending a form and getting some response.

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Create a server
http.createServer( function (req, resp) {
  // Parse the request containing file name
  var pathname = url.parse(req.url).pathname;
  // Print the name of the file for which request is made
  console.log("Request for " + pathname + " received.");

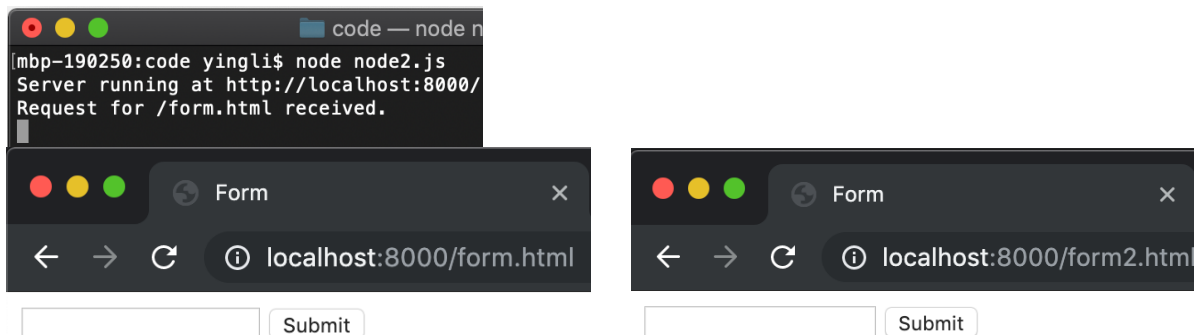
  // Read the requested file content from file system
  fs.readFile(pathname.substr(1), function (err, data) {
    if (err) {
      console.log(err);
      // HTTP Status: 404 : NOT FOUND
      // Content Type: text/plain
      resp.writeHead(404, {'Content-Type': 'text/html'});
    }
    else {
      // Page found
      // HTTP Status: 200 : OK
      // Content Type: text/plain
      resp.writeHead(200, {'Content-Type': 'text/html'});

      // Write the content of the file to response body
      resp.write(data.toString());
    }
    // Send the response body
    resp.end();
  });
}).listen(8000);

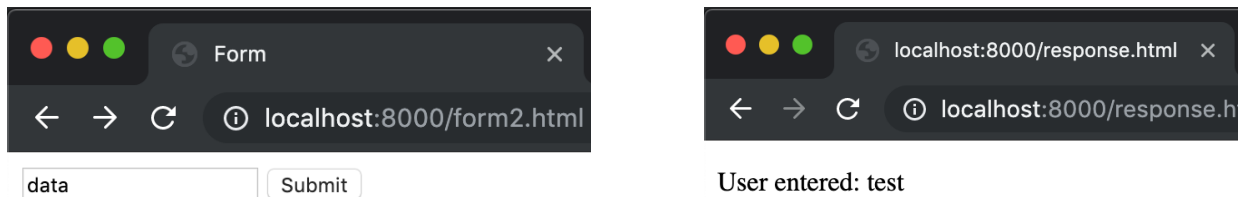
// Console will print the message
console.log('Server running at http://localhost:8000/');
```

- show the code of server2.js
 - the server listens at port 8000
 - response to the request from clients after parsing the request (pathname) and send the requested file back to the client.
- The module fs provides a variety of file system methods.
- The easiest way to read the entire file is using readFile method:
 - fs.readFile(file, [encoding], [callback]);
 - file is a string indicating the path of the file to read

- encoding is an optional parameter that specifies the type of encoding to read the file, such as 'ascii', 'utf8', and 'base64'. The default is null if no encoding is provided.
 - callback is a function to call when the file has been read and the contents are ready. It has two parameters, error and data. If there is no error, error will be null and data will contain the file content. Otherwise, error will contain the error message.
- References: <https://nodejs.org/en/knowledge/file-system/how-to-read-files-in-nodejs/>
- run server2.js and request form1.html and form2.html from web browser



- The server2.js server can send the responses to the web browser according to the requests.
- However, the response page cannot reflect the submitted data correctly. Submitted "data" and return "test".



- Because the action field of the `<form>` in `form1.html` and `form2.html` is a statical html page, which cannot reflect the data submitted by the client.
 - action defines the response after submission



- To make an interactive webpage, there should be a way to transmit data between the client and server back and forth, even better, without impaction to the current page display (no need to reload the page for every submission).

Node.js Example 3

- Client side AJAX
 - send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page.
 - It's common to use GET in AJAX unless sending a sensitive data or calling script processing data on the server

```

<script>
  $("form").submit(function(f) {
    f.preventDefault();
    $.ajax({
      url: "/request",
      data: "q="+$("#q").val(),
      type: "GET",
      success: function(data) {
        var results = JSON.parse(data);
        $("#result").html(results["result"]);
      },
      error: function(e) {
        alert("Error: "+e.status+": "+e.statusText);
      }
    });
  });
</script>

```

GET /request?q=test HTTP/1.1

{"result": "server response"}

- Server side
 - Since we want to display the user entered string in the Results table, we need to fetch the string from the query part of the url, and then send it back with the HTTP response.
 - Once the input is submitted, the pathname becomes "/request." So, we can use it as a trigger to parse the query. If the pathname is not "/request," send the required file back to the client.

```

if (pathname == '/request') {
  var query = url.parse(req.url).query;
  console.log("query " + query + " received.");

  resp.writeHead(200, {'Content-Type': 'text/html'});
  resp.write('{"result":"' + query.substring(2) + '"}');
  resp.end();
}

```

- Run server3.js and open form3.html on web browser.

Node.js Example 3 Extension

- In this specific example, the server doesn't need to do anymore computation but send the query string back to the client. However, sometime, the server side needs to do calculate the results and then send the results back. E.g., the server may need to search into a database or run some program to calculate the results. Then, we can let the server spawn a child process to do the calculation and then send the result back when the calculation is done.
- Using the same example, we can let the server executes an echo command that prints the stdin string to the stdout. The server echos the query, and once it's done, send the stdout string to the client side.

```
if (pathname == '/request') {  
  var query = url.parse(req.url).query;  
  console.log("query " + query + " received.");  
  
  // Way 2: generate a child process to produce data and send back  
  var spawn = require('child_process').spawn;  
  var cp = spawn('echo', ['{"result":"' + query.substring(2) + '"}']);  
  cp.stdout.on('data', function(data) {  
    console.log(`child process stdout:\n${data}`);  
    resp.writeHead(200, {'Content-Type': 'text/html'});  
    resp.write(data.toString());  
    resp.end();  
  });  
}
```

Disclaimer: Notes adapted from the textbook and online resources.