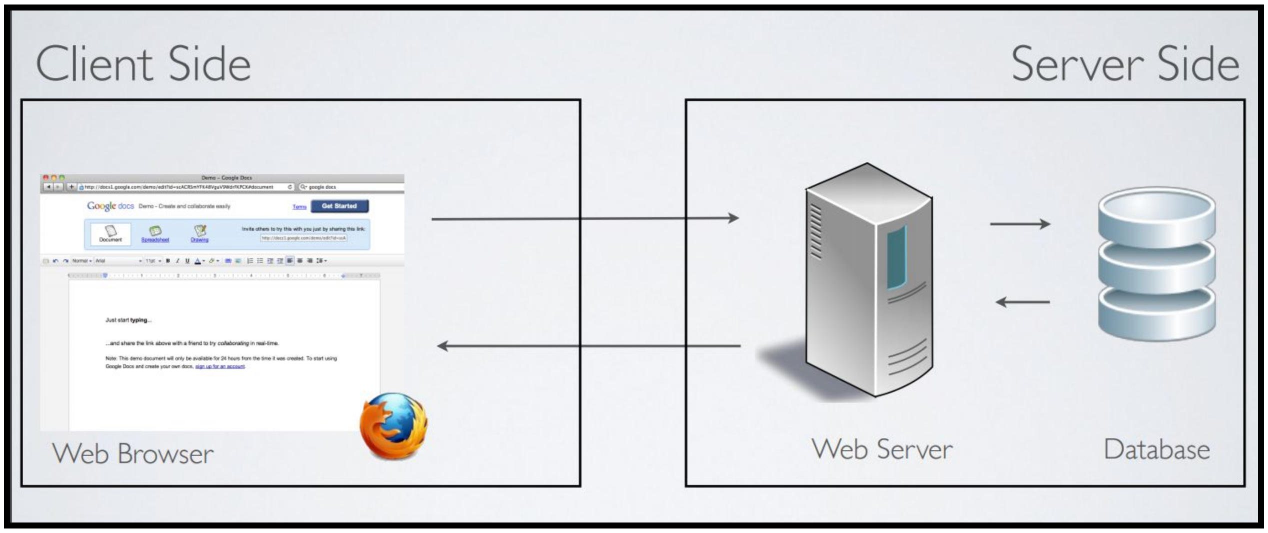
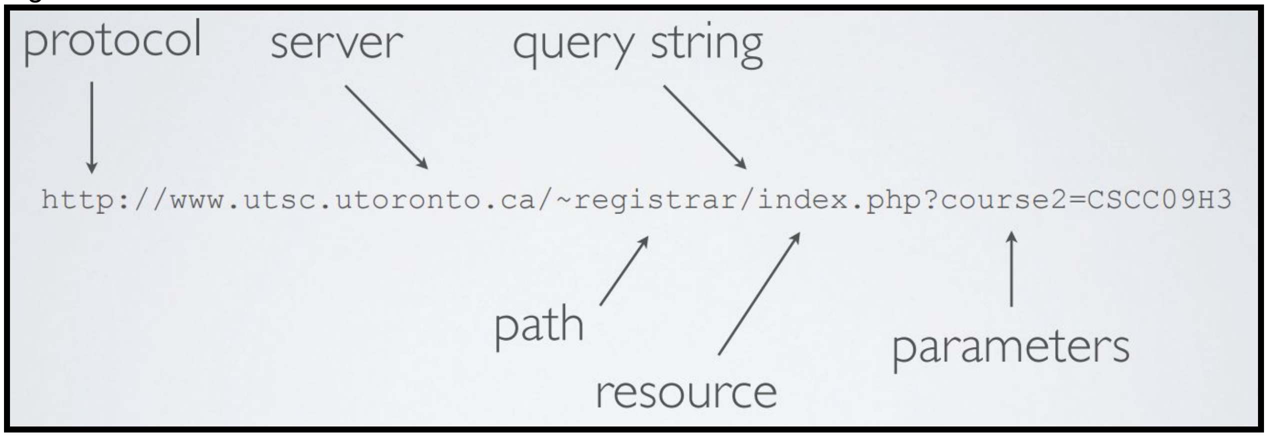
HTTP Protocol:

- **Introduction:**
- HTTP stands for **HyperText Transfer Protocol**.
- The HTTP protocol is a network protocol for requesting/receiving data on the Web.
- A **client** is usually a web browser.
  A **server** is a machine that hosts the website.



- Communication between clients and servers is done by **HTTP requests** and **HTTP responses** like such:
  1. A client sends an HTTP request to the web.
  2. A web server receives the request.
  3. The server runs an application to process the request.
  4. The server returns an HTTP response to the browser.
  5. The client receives the response.
- The server usually runs the standard TCP protocol on port 80 by default.
  I.e. The browser connects to the server on port 80 by default.
  Note: Port 80 is used if we're using HTTP. If we're using HTTPS, the port is 443.
- The **URI (Uniform Resource Identifier)/URL (Uniform Resource Locator)** specifies what resource is being accessed.
- A **Uniform Resource Identifier (URI)** is a unique identifier used by web technologies. URIs may be used to identify anything, including real-world objects, such as people and places, concepts, or information resources such web pages and books. Some URIs provide a means of locating and retrieving information resources on a network (either on the Internet or on another private network, such as a computer filesystem or an Intranet), these are **Uniform Resource Locators (URL)**.
  I.e. A URI is an identifier of a specific resource while a URL is a special type of URI that also tells you how to access it.

- **Anatomy of a URL:**
- E.g.


protocol
server
query string
http://www.utsc.utoronto.ca/~registrar/index.php?course2=CSCC09H3
path
resource
parameters

- HTTP is the protocol.
- www.utsc.utoronto.ca is the name of the server.
- Then, you have the path which refers to a resource in the webserver.
  Note: Sometimes, you might not see the resource in the URL.
- Then we have a question mark (?). This question mark represents the start of **query parameter(s)**. Query parameters are usually key-value pairs.
  Note: If there are multiple query parameters, they are separated by an ampersand (&).
  E.g. Consider this URL:
  http://www.google.co.uk/search?**q=url&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-GB:official&client=firefox-a**
  Notice that each query parameter is separated by a &.
- **Note:** The query string contains the path, resource and parameters.
- **HTTP Request Methods:**

Request Method	**Explanation**
GET	Requests a representation of the specified resource. Requests using GET should only retrieve data.
POST	Used to submit an entity to the specified resource, often causing a change in state or side effects on the server. I.e. Adds an unidentified resource.
PATCH	Used to update to a resource.
PUT	Adds an identified resource. The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly has side effects of creating the same resource multiple times.
DELETE	Deletes the specified resource.

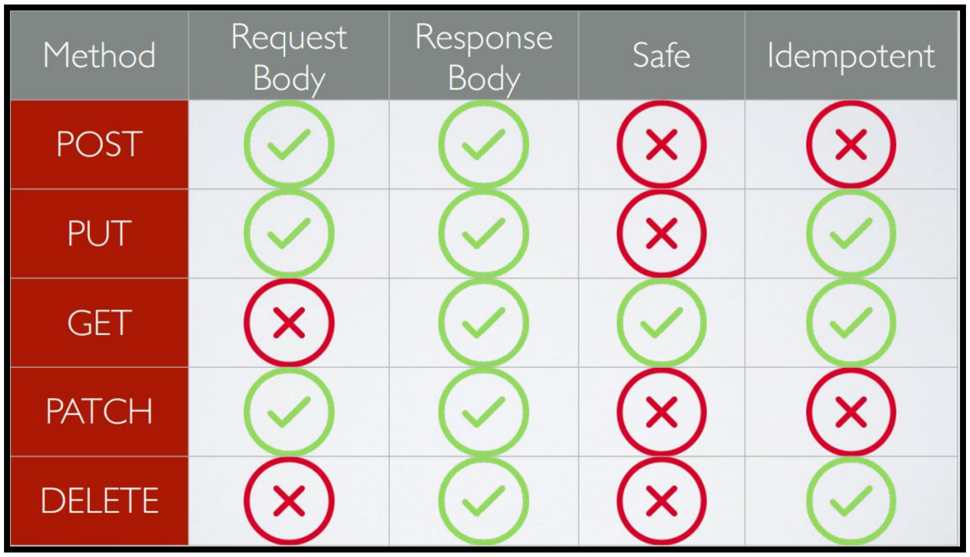
- **HTTP Request:**
- An HTTP request contains the following information:
    - A request method.
    - A query string.
    - A header, which is a key-value pair.
    - An optional body which contains data.
- **Curl Command:**
- We can use the curl command to send HTTP requests to a server.
- Syntax: **curl options url**

  options include:

- **-v**	*verbose*
- **--request**	*request_method*
- **--data**	*request_body*
- **--header**	*header*

- **HTTP Response:**
- An HTTP response contains the following information:
    - A status code.
    - A header, which is a key-value pair.
    - An optional body which contains data.
- **Status Codes:**
- 1xx - information
- 2xx - success
- 3xx - redirection
- 4xx - client error

  I.e. The client tries to find a web page that doesn't exist.
- 5xx - server errors
- **Method Properties:**
- An HTTP request/response:
    - May have a request body.
    - May have a response body.
    - May not have side effects. **(Safe)**
    - May have the same result when called multiple times. **(Idempotent)**
- The developer/programmer makes the choice.
- What the standard recommends

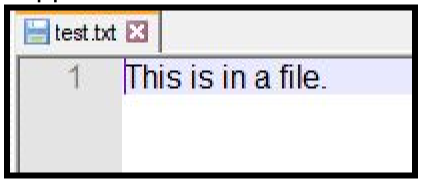


Method	Request Body	Response Body	Safe	Idempotent
POST	✓	✓	✗	✗
PUT	✓	✓	✗	✓
GET	✗	✓	✓	✓
PATCH	✓	✓	✗	✗
DELETE	✗	✓	✗	✓

- **Difference Between HTTP and HTTPS:**
- HTTP is unsecure while HTTPS is secure.
- HTTP sends data over port 80 while HTTPS uses port 443.
- HTTP operates at the application layer, while HTTPS operates at the transport layer.
- No SSL certificates are required for HTTP, but it is required that you have an SSL certificate and it is signed by a CA for HTTPS.
- HTTP doesn't require domain validation, whereas HTTPS requires at least domain validation and certain certificates even require legal document validation.
- There is no encryption in HTTP, whereas with HTTPS the data is encrypted before sending.

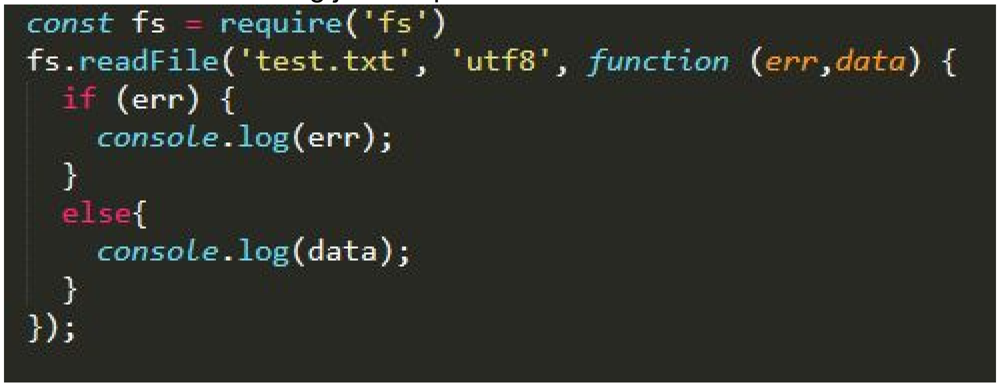
JavaScript on the Server:

- We'll be using Node.js for this class.
- Node.js:
  - Runs on Chrome V8 Javascript engine.
  - Has non blocking-IO (**asynchronous/event-driven**).
  - Has no restrictions unlike when js is running on the browser.
- E.g. of a node.js file reading another file:
  Suppose we had a file called "test.txt" with the line "This is in a file." in it.

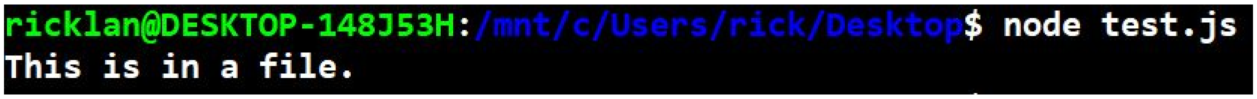


```
This is in a file.
```

We can use the following javascript code to read it.



```
const fs = require('fs')
fs.readFile('test.txt', 'utf8', function (err,data) {
  if (err) {
    console.log(err);
  }
  else{
    console.log(data);
  }
});
```

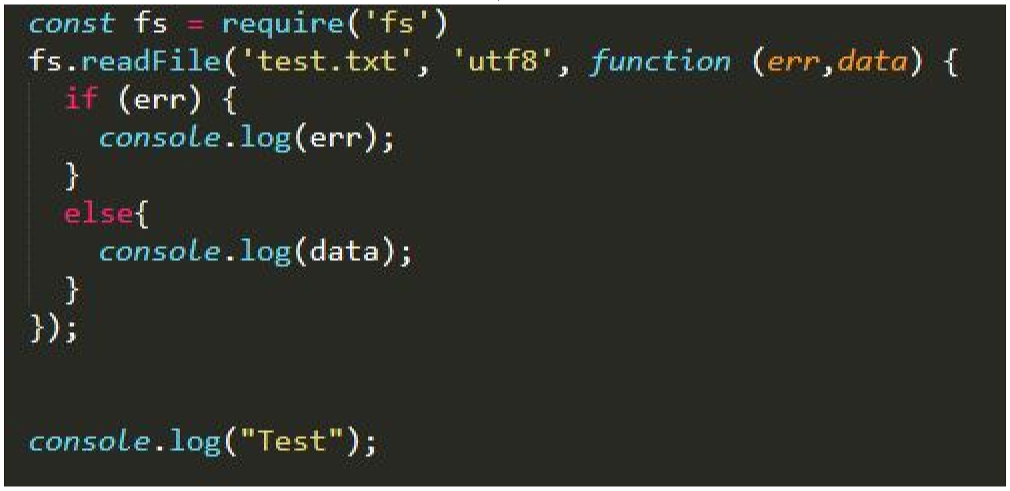


```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ node test.js
This is in a file.
```

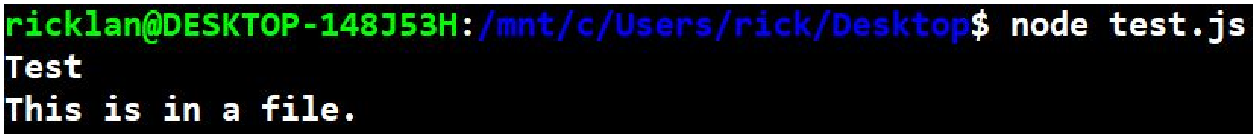
Note: We couldn't use JavaScript on the browser to read files on a user's computer.

- **Blocking code** is code that runs one instruction after another, and makes next instructions wait.
- **Non-blocking code** means that we don't have to wait for some code to run before running other code.
  I.e. Blocking refers to operations that block further execution until that operation finishes while non-blocking refers to code that doesn't block execution.
- Blocking methods execute synchronously and non-blocking methods execute asynchronously.

- All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with Sync.
- E.g. of non-blocking I/O:
  We'll use the same .txt file from above, but make one small modification to the .js file.

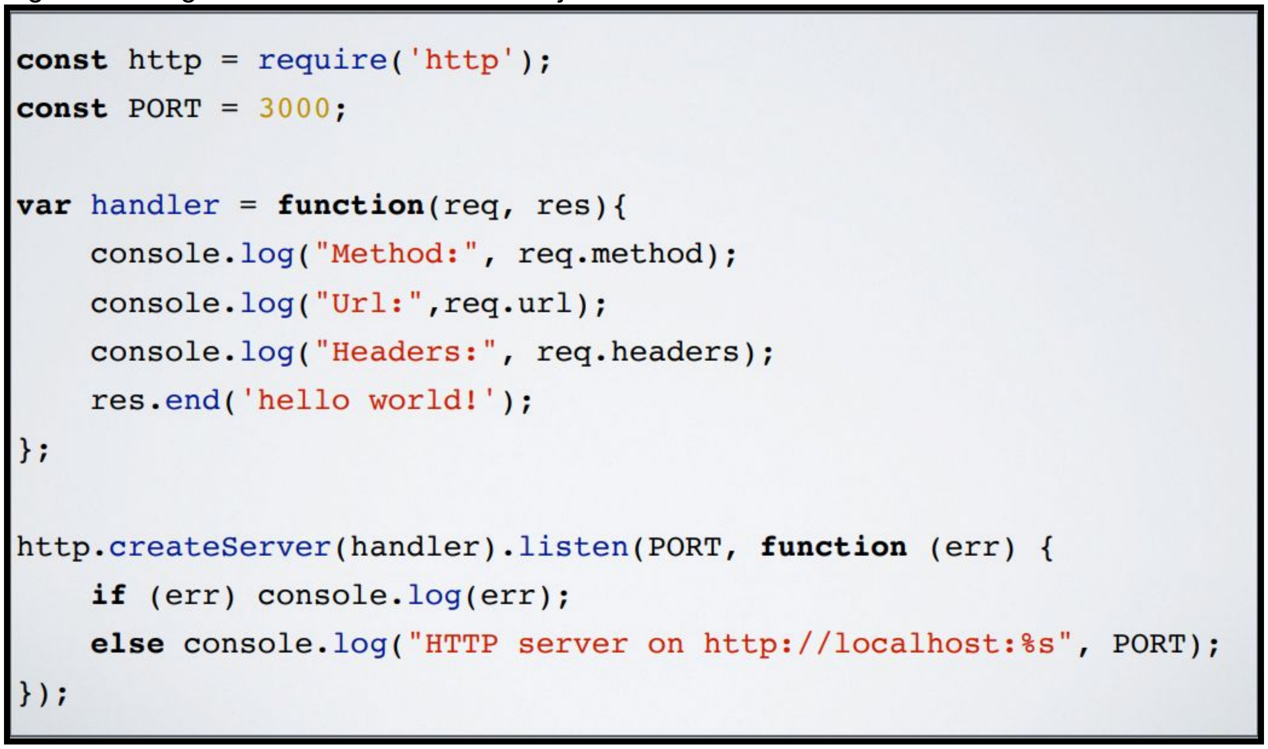


```
const fs = require('fs')
fs.readFile('test.txt', 'utf8', function (err,data) {
  if (err) {
    console.log(err);
  }
  else{
    console.log(data);
  }
});


console.log("Test");
```



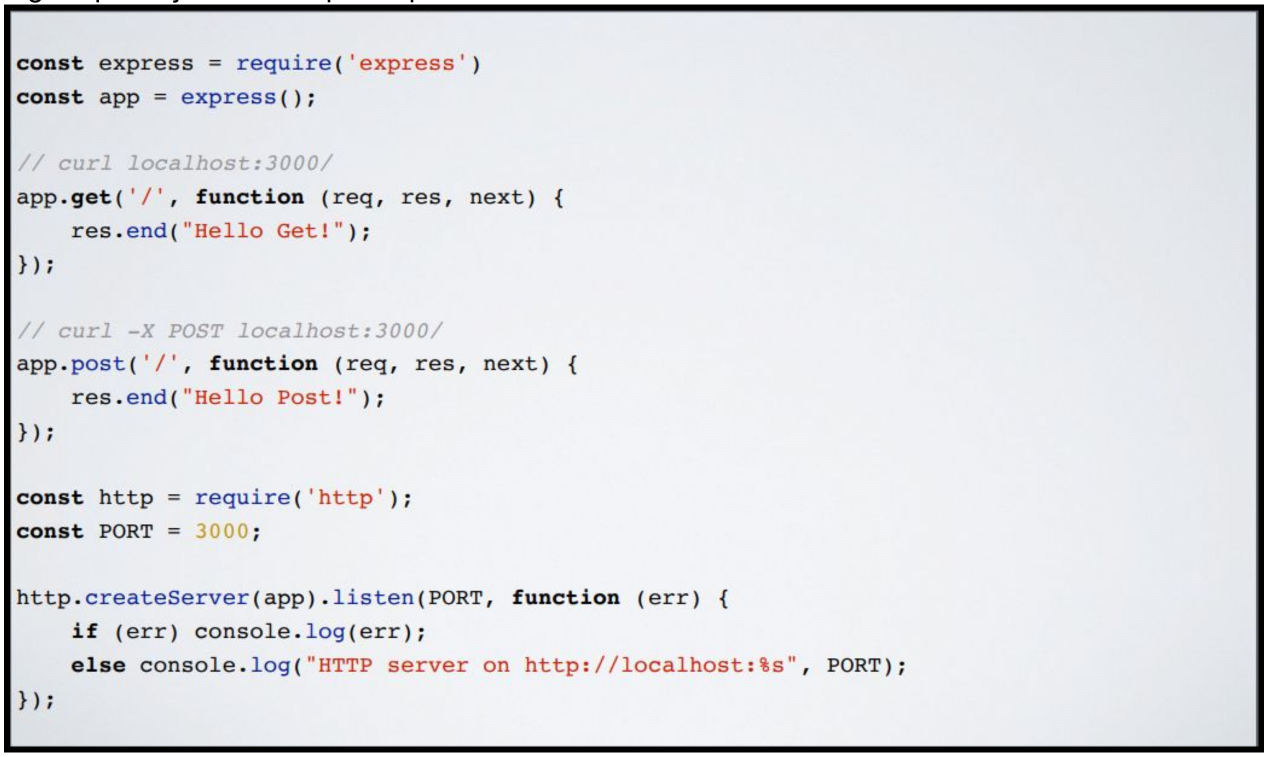
```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ node test.js
Test
This is in a file.
```

  Notice that even though console.log("Test"); was after the readFile section, it was printed first. This is because of non-blocking I/O.
  Note: The blocking version of readFile is readFileSync.
- E.g. of Building an HTTP server with Node.js:



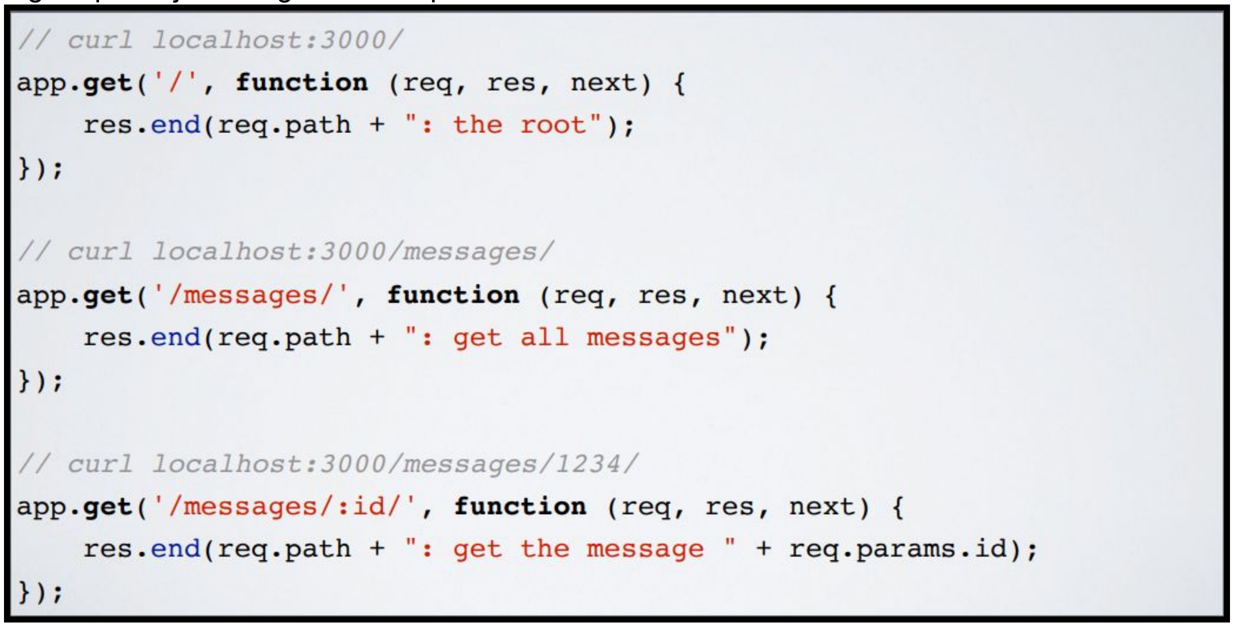
```
const http = require('http');
const PORT = 3000;

var handler = function(req, res){
    console.log("Method:", req.method);
    console.log("Url:",req.url);
    console.log("Headers:", req.headers);
    res.end('hello world!');
};

http.createServer(handler).listen(PORT, function (err) {
    if (err) console.log(err);
    else console.log("HTTP server on http://localhost:%s", PORT);
});
```

- We need to process HTTP requests and execute different actions based on:
  - The request method
  - The url path
  - Whether the user is authenticated
- A router can be written from scratch but it is tedious. Instead, we'll use the backend framework Express.js.
  Note: Express is not part of the default library in Node.js. We have to install it.
- E.g. Express.js with multiple request methods:



```
const express = require('express')
const app = express();

// curl localhost:3000/
app.get('/', function (req, res, next) {
    res.end("Hello Get!");
});

// curl -X POST localhost:3000/
app.post('/', function (req, res, next) {
    res.end("Hello Post!");
});

const http = require('http');
const PORT = 3000;

http.createServer(app).listen(PORT, function (err) {
    if (err) console.log(err);
    else console.log("HTTP server on http://localhost:%s", PORT);
});
```

- E.g. Express.js routing based on path:

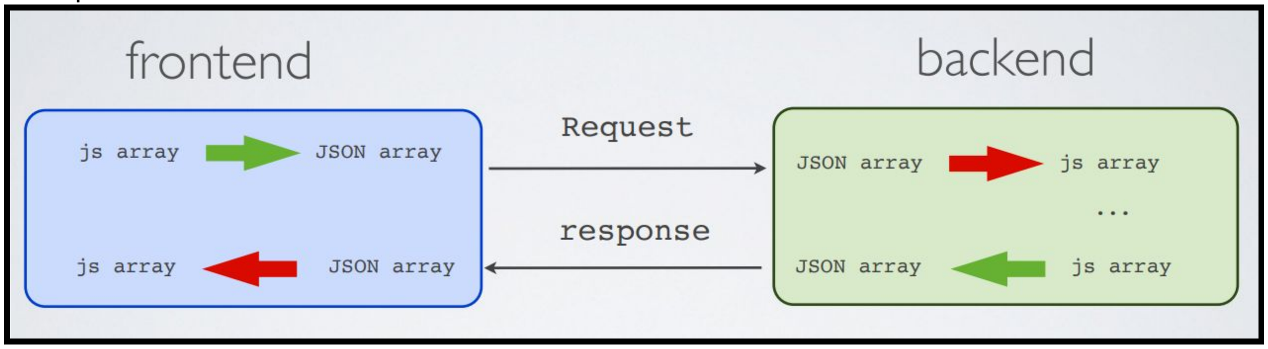


```
// curl localhost:3000/
app.get('/', function (req, res, next) {
    res.end(req.path + ": the root");
});

// curl localhost:3000/messages/
app.get('/messages/', function (req, res, next) {
    res.end(req.path + ": get all messages");
});

// curl localhost:3000/messages/1234/
app.get('/messages/:id/', function (req, res, next) {
    res.end(req.path + ": get the message " + req.params.id);
});
```

- The body of HTTP request and response is a string. A problem arises if we want to pass data structures. 3 possible solutions are encoding it with either:
    1. URI encoding (sometimes used)
    2. XML encoding (rarely used these days)
    3. JSON encoding (very frequently used these days)
- By default, the frontend and backend uses URI encoding. If you want to encode using JSON, you need to tell the frontend/backend that you're using JSON. To do it, you can use "application/json".

JSON:

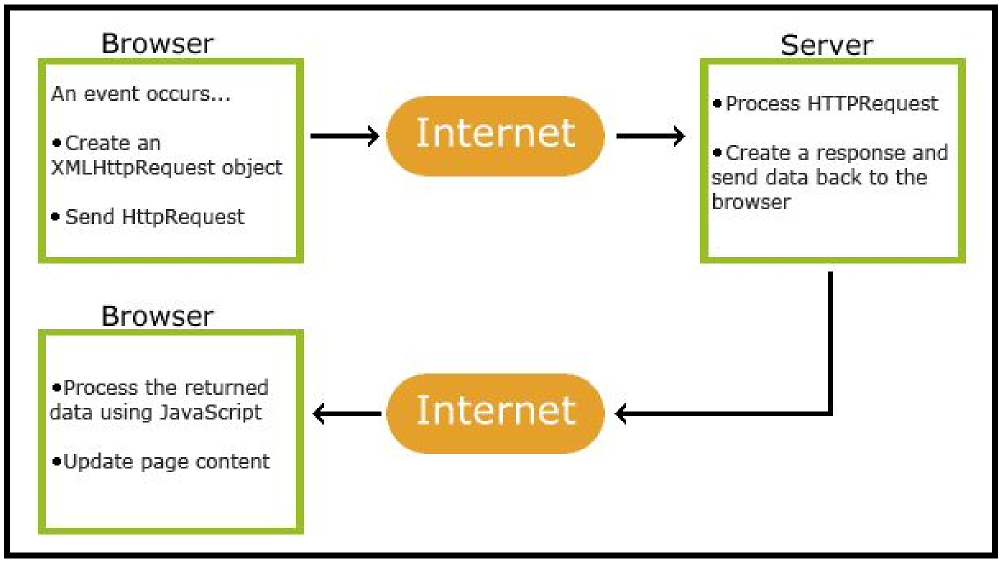
- Stands for **JavaScript Object Notation**.
- Originally, we used URI encoding, but it was hard to send data structures. Then, we used XML, but it's very hard for people to read. Then, someone came up with JSON.
- It is a way to encode data structures into a string. JSON is not a language.
- JSON is a human readable lightweight open format to interchange data.
- Since 2009 browsers support JSON natively.
- A JSON data structure is either:
    1. array (indexed array)
    2. object (associative array)
- JSON values are:
    1. string
    2. number
    3. true
    4. false
    5. null
- **Serialization** means that we're converting JavaScript into JSON.
  I.e. **var myJSONText = JSON.stringify(myObject);**
- **Deserialization** means that we're converting JSON into JavaScript.
  I.e. **var myObject = JSON.parse(myJSONtext);**
- Example of serialization and deserialization:



Ajax:

- AJAX is not a programming language. It is a simple JavaScript command.
- AJAX is a technique for accessing web servers from a web page without refreshing the page.
  I.e. AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.
- AJAX stands for **Asynchronous JavaScript And XML**.
- AJAX revolutionized the web. It started with Gmail and Google Maps.

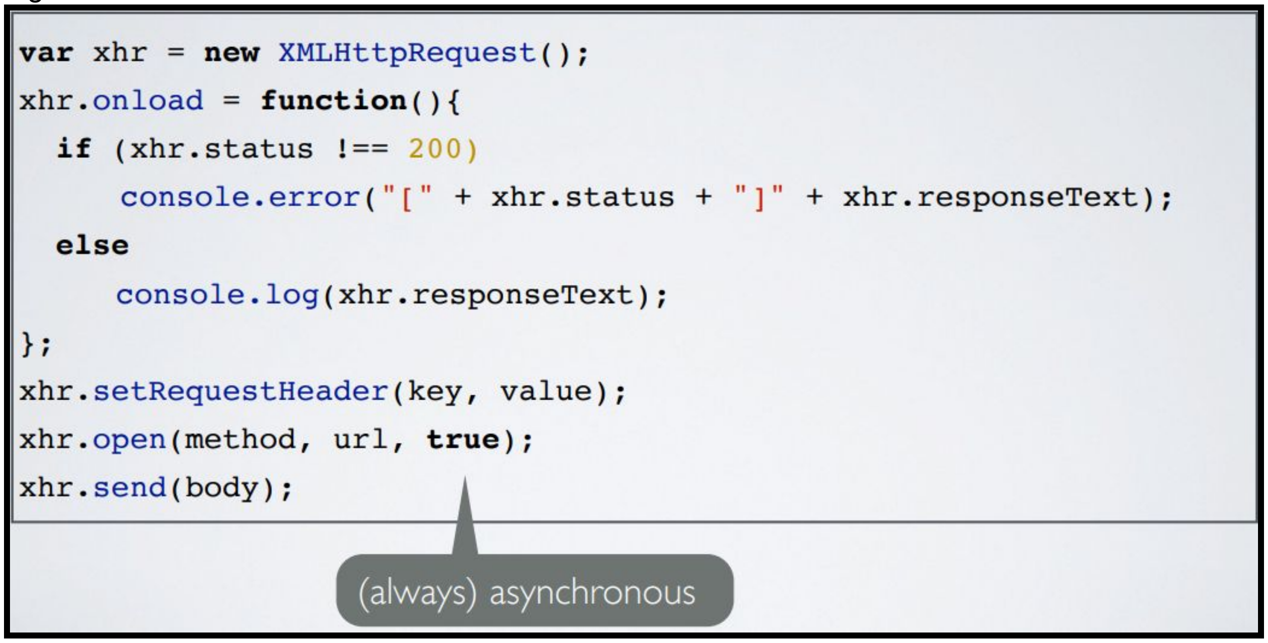
- Advantages:
    1. Low latency
    2. Rich interactions
- Consequences:
    1. The webapp center of gravity moved to the client side.
    2. Javascript engine performance race.
- AJAX just uses a combination of:
    1. A browser built-in XMLHttpRequest object to request data from a web server.
    2. JavaScript and HTML DOM to display or use the data.
- How AJAX works:


Browser
An event occurs...
•Create an XMLHttpRequest object
•Send HttpRequest
Internet
Server
•Process HTTPRequest
•Create a response and send data back to the browser
Internet
Browser
•Process the returned data using JavaScript
•Update page content

1. An event occurs in a web page.
2. An XMLHttpRequest object is created by JavaScript.
3. The XMLHttpRequest object sends a request to a web server.
4. The server processes the request.
5. The server sends a response back to the web page.
6. The response is read by JavaScript.
7. A proper action, like page update, is performed by JavaScript.

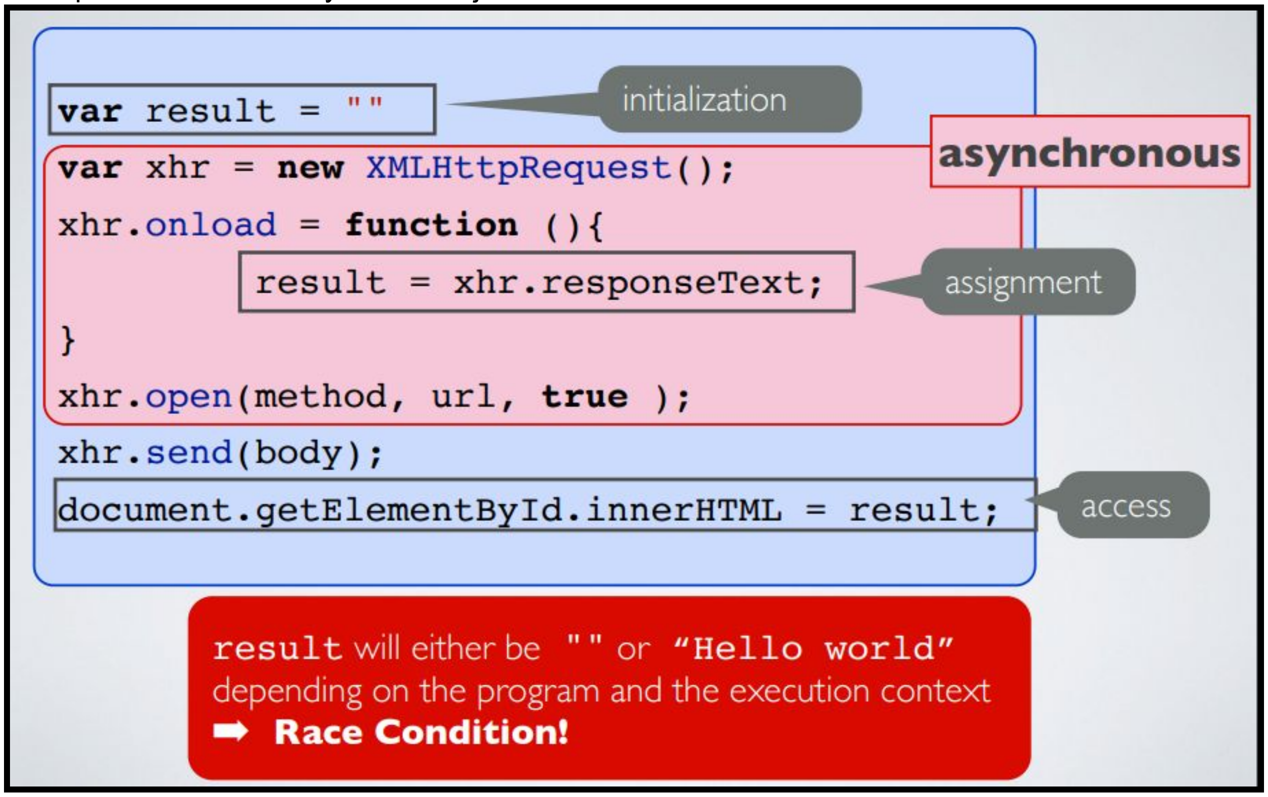
- To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object.
  xhttp.open(method, text, true);
  xhttp.send() or xhttp.send(string);
  Note: Use **xhttp.send()** for GET and **xhttp.send(string)** for POST.
- Server requests should be sent asynchronously. This means that the async parameter of the open() method should be set to true.
- By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:
    1. Execute other scripts while waiting for server response.
    2. Deal with the response after the response is ready.

E.g.



```
var xhr = new XMLHttpRequest();
xhr.onload = function(){
  if (xhr.status !== 200)
      console.error("[" + xhr.status + "]" + xhr.responseText);
  else
      console.log(xhr.responseText);
};
xhr.setRequestHeader(key, value);
xhr.open(method, url, true);
xhr.send(body);
```

- Concurrency is a big issue in Ajax.
- Example of a concurrency issue in Ajax.

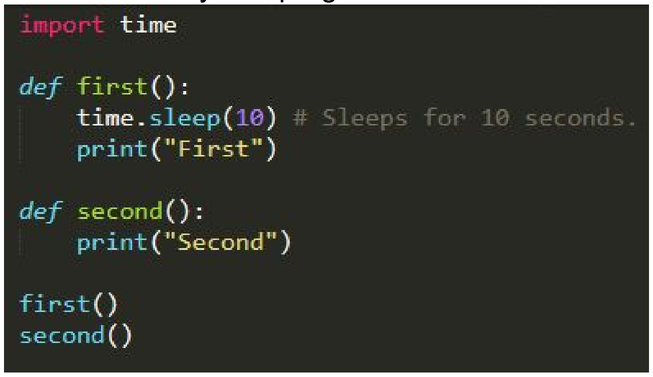


```
var result = ""
var xhr = new XMLHttpRequest();
xhr.onload = function (){
        result = xhr.responseText;
}
xhr.open(method, url, true );
xhr.send(body);
document.getElementById.innerHTML = result;
```

`result` will either be `""` or `“Hello world”` depending on the program and the execution context
➡ **Race Condition!**

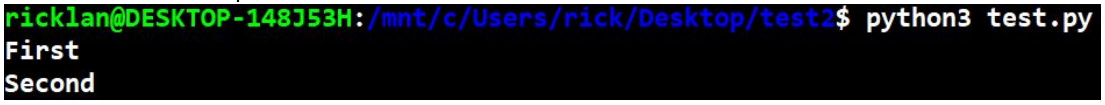
Callback:

- In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called **higher-order functions**. Any function that is passed as an argument is called a **callback function**.
- Callbacks are a way to make sure certain code doesn't execute until other code has already finished execution.
  I.e. Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.
- The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.
- **Note:** The callback function is helpful when you have to wait for a result that takes time. For example, the data coming from a server because it takes time for data to arrive.
- Since JavaScript is an event driven language, we can use callback functions for event declarations, too.
- Example that JavaScript is asynchronous.
  Consider this Python program:



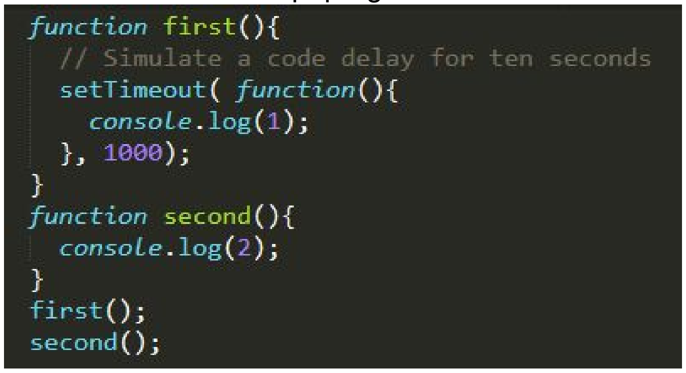
```python
import time

def first():
    time.sleep(10) # Sleeps for 10 seconds.
    print("First")

def second():
    print("Second")

first()
second()
```

When you run it, it waits for 10 seconds before printing First and then prints Second.
I.e. It runs in a sequential order.



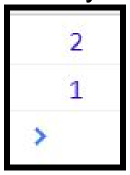
```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/test2$ python3 test.py
First
Second
```

Consider this JavaScript program that does the same thing:



```javascript
function first(){
  // Simulate a code delay for ten seconds
  setTimeout( function(){
    console.log(1);
  }, 1000);
}
function second(){
  console.log(2);
}
first();
second();
```

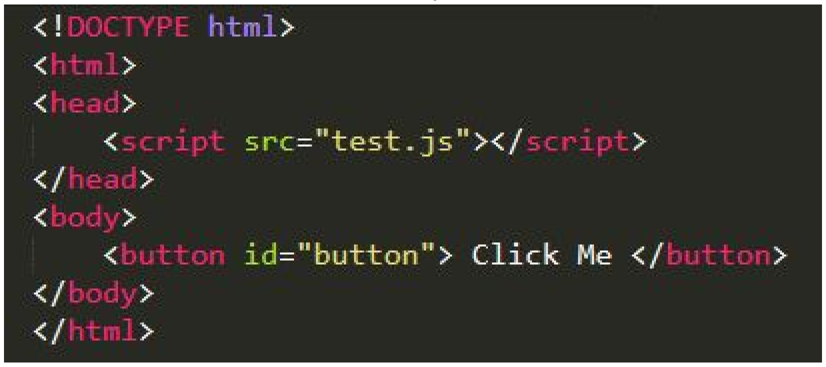
When you run the program, even though you call first() first, 2 will be printed first.



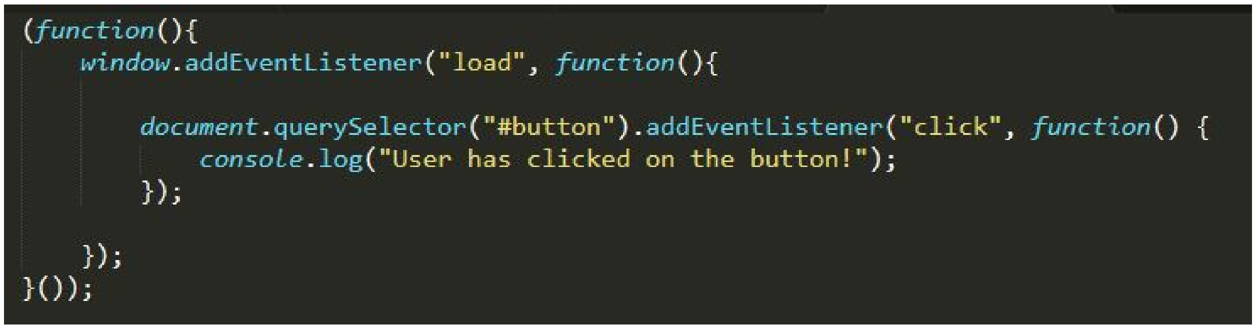
```
2
1
>
```

This is because JavaScript is asynchronous.
Note: In the above JavaScript file, the function in setTimeout is an example of a callback function.

- Example of callback functions used in events.
  Consider the HTML and JavaScript code below:



```html
<!DOCTYPE html>
<html>
<head>
    <script src="test.js"></script>
</head>
<body>
    <button id="button"> Click Me </button>
</body>
</html>
```



```javascript
(function(){
    window.addEventListener("load", function(){

        document.querySelector("#button").addEventListener("click", function() {
            console.log("User has clicked on the button!");
        });

    });
}());
```

We have 2 callback functions in the JavaScript code.
The first is in window.addEventListener and the second is in document.querySelector("#button").addEventListener.
The functions won't run unless the event is triggered.