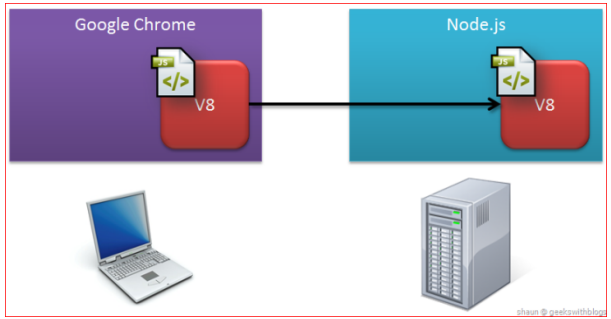
***CS1320***
***Creating Modern Web and Mobile Applications***

Lecture 14

# Web Application Architectures I

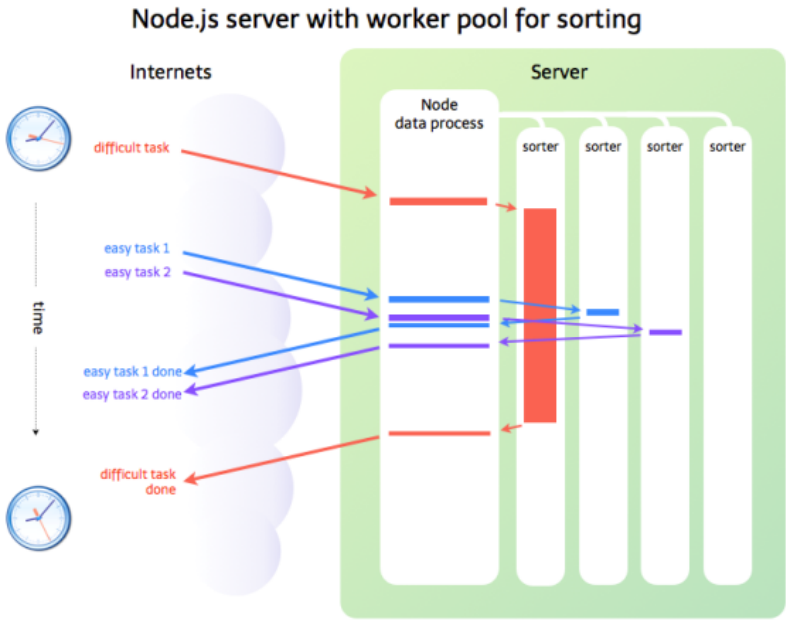
# Events in Node.JS




- Recall our server game
  - Multiple people help speed up the service
  - Multitasking can speed up the service
- How to achieve multitasking?
  - Multiple threads
    - This is what apache, nginx, tomcat, … do
    - Threaded coding can be very complex
    - JavaScript does not support threads
  - Multiple servers
    - Need to ensure same user gets the same server
    - Supported by nginx directly
    - Supported by various front ends for apache
    - Supported by a node.js plug-in
  - Multitask without threads

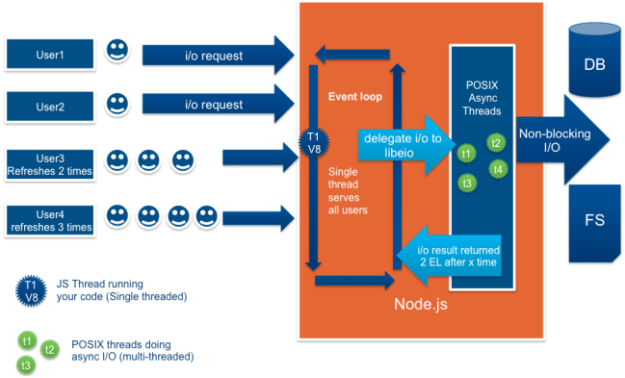
# Events in Node.JS

- What does the web server spend its time doin
  - Listening for requests
  - Reading/writing from the network and files
  - Accessing a database or outside server
  - Not much time is spent doing computation
- These tasks run elsewhere
  - Done in the operating system
  - Done in database system or application server
  - Done in background threads in node.js (not javascript)
  - The web server for an app proper spends its time waiting
- Rather than waiting, use non-blocking I/O
  - Start the I/O and let someone else run
  - When I/O finishes, the server is notified and it processes the result
  - Multiple I/O operations can be pending at once
  - Other operations can be treated as I/O

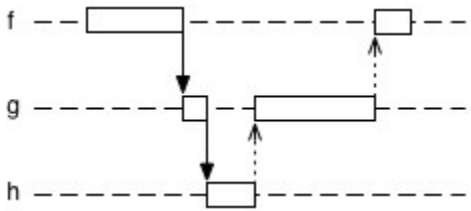


# Events and Event Handlers

- Recall how JavaScript works in the browser
  - JavaScript registers for events (onXXX='function()')
  - When something happens, JavaScript is invoked to change the DOM
  - The browser continues execution when JS returns
    - And the change is effected
- Node.JS takes this approach
  - Start an operation via a function call
    - Operation defines a set of events tagged by name
    - Register callbacks (functions) for events of interest
    - Return control to Node.JS
    - This is when the operation actually begins
  - Node.JS will run the operation in background
    - Invoke your callback functions as needed



# Functions and Continuations



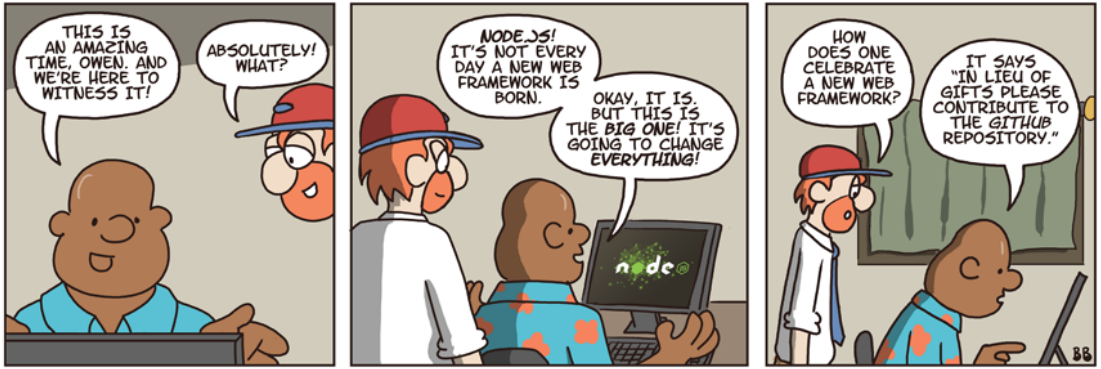

- Callbacks are functions in JavaScript
  - Arguments determined by the event
- Functions in JavaScript can be defined in-line

```
db.query("…",[…],function (e1,d1) { hQ2(req,res,e1,d1); } );
db.query("…",[…], (e1,d1) => { hQ2(req,res,e1,d1); } );
```

  - When a function is defined this way
    - It can access variables/parameters of the outer function
  - This is effectively a ***continuation***
    - I.e. the inner function defines how execution should continue
    - When the specific event occurs
- Coding practice
  - **Do as multiple functions (very simple in-line function calling next)**
  - **Or use Promises with functions defined separately (not nested)**

# Node.JS Modules

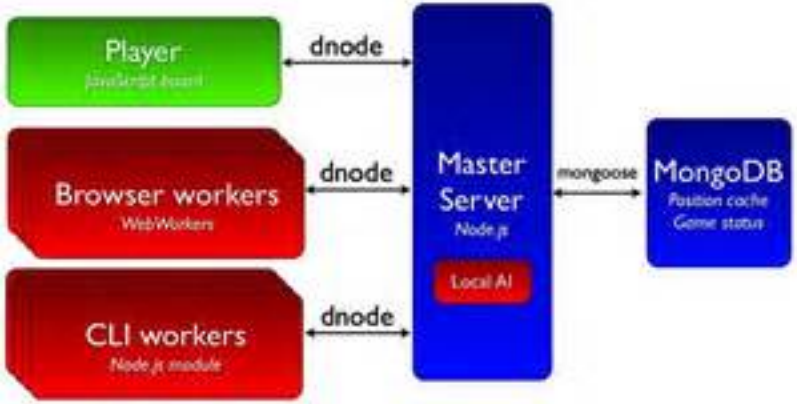
- Synchronous
  - URL decoding
  - File path manipulations
  - Assertions, debugging, read-eval-print loop
  - OS queries
  - Utilities
- Plus external module




Not Invented Here™ © Bill Barnes & Paul Southworth NotInventedHere.com

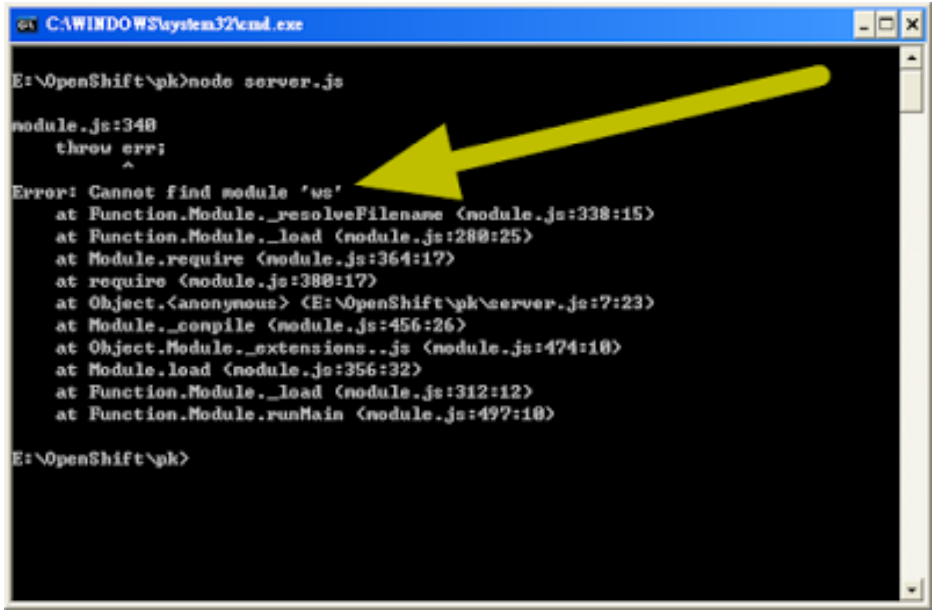
# Node.JS Modules

- Asynchronous (event-based)
  - File I/O
  - External processes and code (C/C++)
  - HTTP, HTTPS
  - Crypto, TLS/SSL
  - Database access (SQL/MANGO)
  - Timers
  - Web sockets
- Plus external modules



# Node.JS Weaknesses

- Documentation
- Coding errors
- Error Recovery
- Scalability

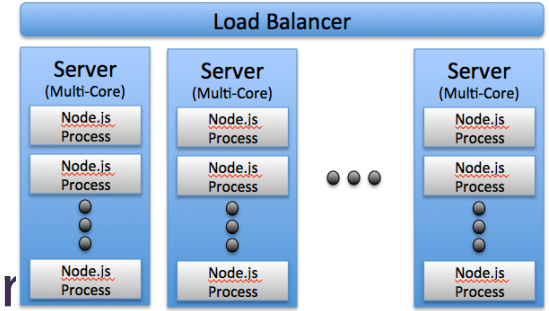


# Node.JS Error Recovery

- Node.JS (your server) will halt:
  - At start up if the JavaScript doesn't compile
  - At run time if there are any run time errors
- Is this the desired behavior?
- Exceptions, try … catch
  - Doesn't work that well with asynchronous calls
  - What do you do with an exception?
  - Promise.catch
- Domains
  - Provide a more general mechanism
  - Still require considerable coding
- Add error checking code at each stage
- Try to anticipate errors as much as possible
- Express has some error handling modules

```
75
76
77
78  <script type="text/javascript">
79  $.ajax({
80      type: "GET",
81      url: "http://www.free.in/realtimefastindex.ashx?listid=22992273&ac
82      dataType: "jsonp",
83      crossDomain: true,
84      success: function (data) {
85          alert(data.result);
86      },
87      error: function (result, sts, err) {
88          alert(err + " : " + sts);
89      }
90  });
91  </script>
92
93  </body>
```

# Scaling Node.JS

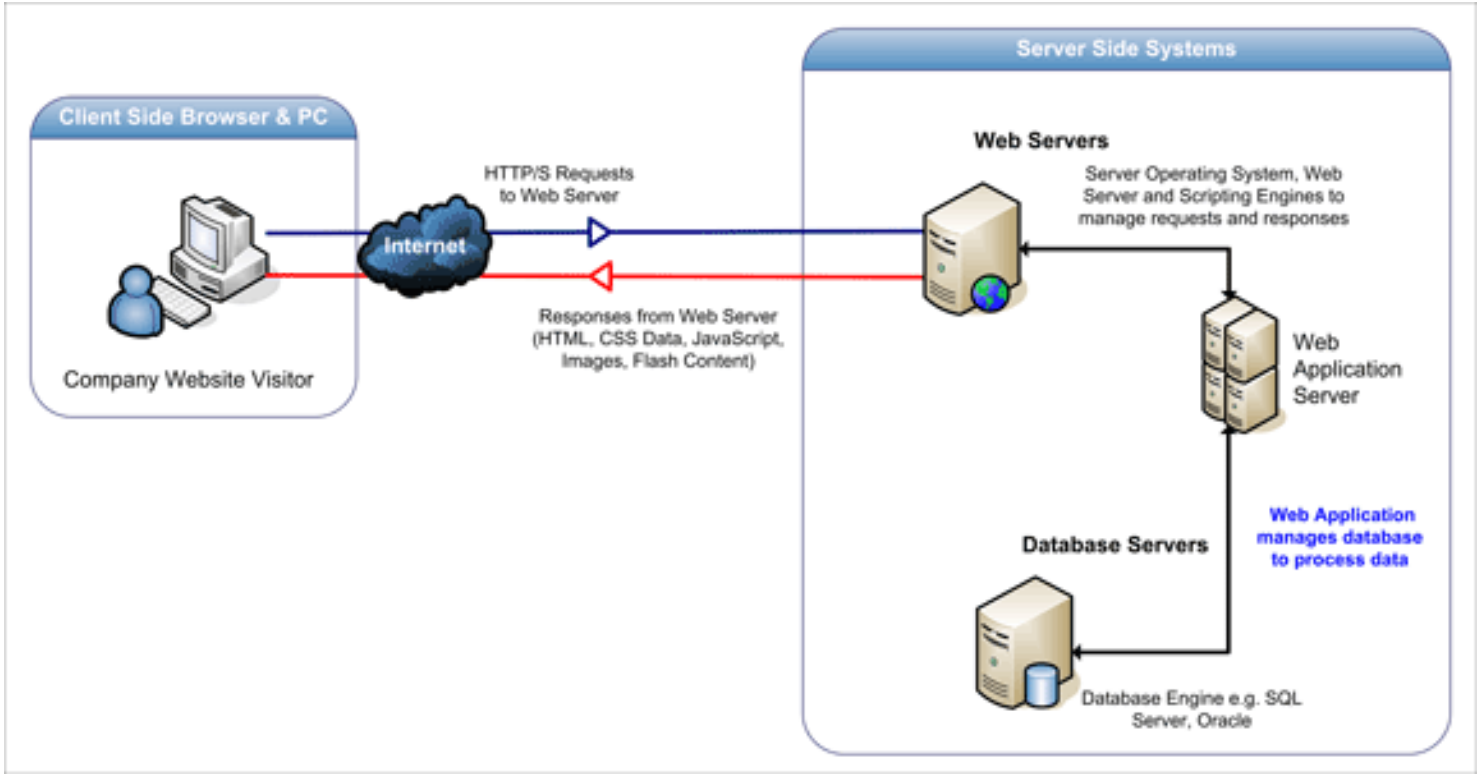



- Requires running multiple Node.JS server
  - On the same machine (multiple cores)
  - On separate machines (cluster)
- And sending requests based on incoming IP address
- Can be done using NginX or other front end
- Can be done using Node.JS
  - There's a module for that

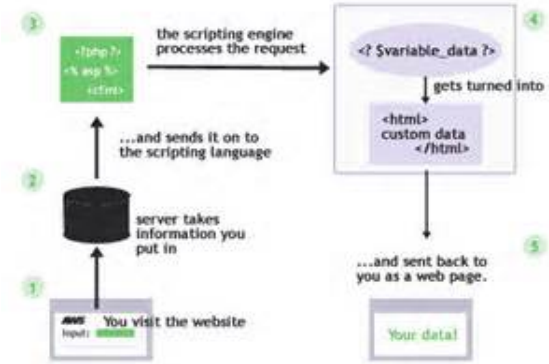
# Web Applications


- Are distributed systems
  - Some work is done in the front end
  - Some work is done in the back end
  - Some work is done in servers or databases
- Different web applications allocate the work differently
  - **Server-side** heavy: banner, blogs, …
  - **Client-side** heavy: gmail, google docs
- What should be done where depends on lots of factors
  - Responsiveness; Performance
  - Access to and security of code and data
  - Amount of communications needed
  - Where the data is actually needed; what is done with the data

# Server-Side Application



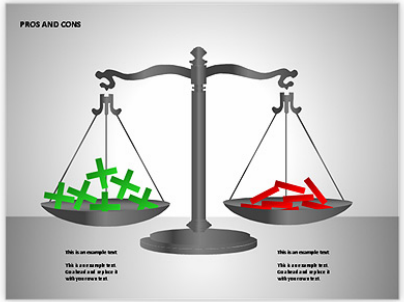
# Server-Side Application



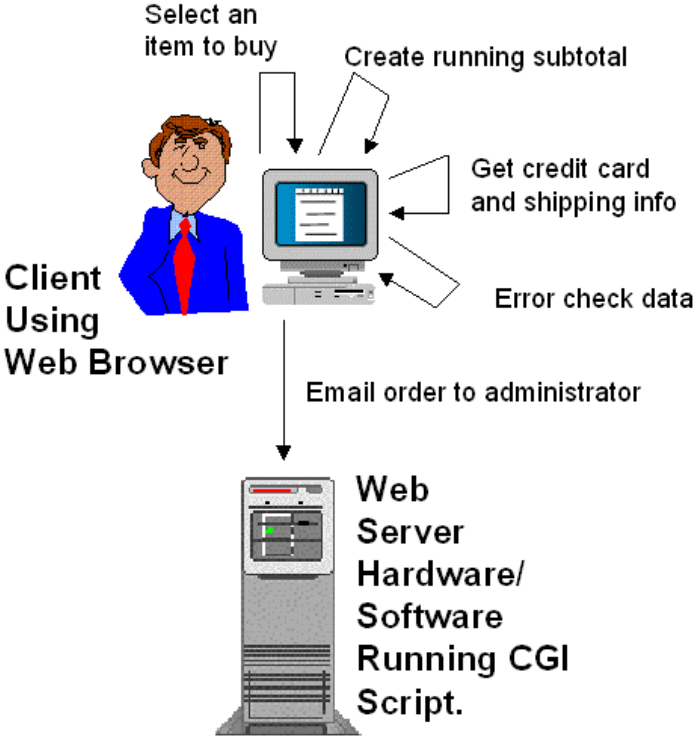

- The default browser-server model
  - Browser sends a HTTP request
  - The HTTP response is used to replace the current page
- Various technologies can support this model
  - Using PHP, JSP, Servlets to generate new HTML page
    - Based on properties passed in the URL query
  - Using Node.JS with a templating engine
  - Front-end JavaScript only used for interactive features (i.e. pull downs, validation)

# Server-Side Pros/Cons

- Templating lets you write HTML directly for the most part
  - Easier to change templates than actual code
- Don't have to send lots of code over the web
  - The code can be kept private
- Server code is generally synchronous, straight-forward
- Data isn't directly accessible to users
- Not as interactive, responsive
- Requires more compute power on server
  - Less on the clients
- Works naturally with assistive devices



# Client Side Application



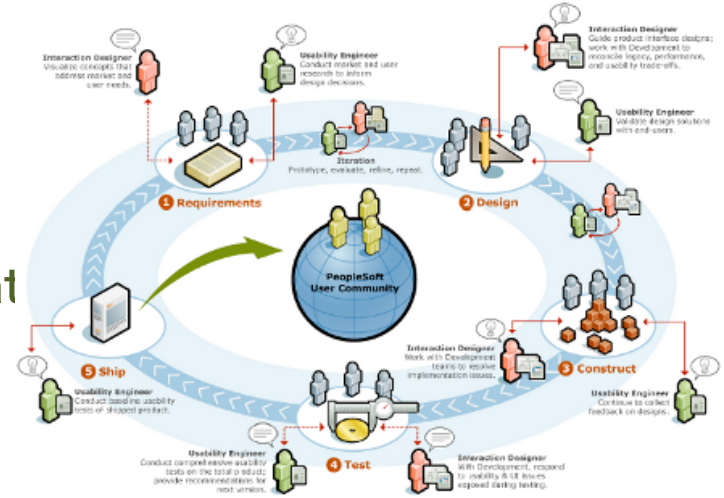

ANGULARJS by Google

React JS

Vue.js

# Client-Side Application

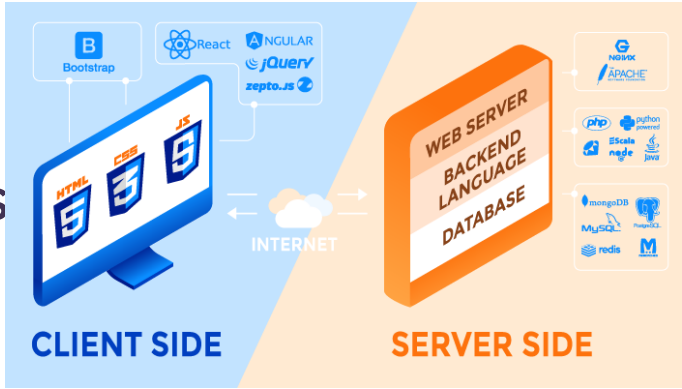
- Most of the work is done on the page
  - Using JavaScript
  - As with Vue, React, Angular, …
- Front end still needs to get/send informat
  - To the server, database, back end, application
  - To actually get work done
  - To get additional information
  - To ensure information is permanent
  - To save status in case of refresh, return to page
- Page update done in JavaScript
  - Based on information retrieved
  - JavaScript handles formatting, updating, etc. the page



# Client-Side Pros and Cons

- JavaScript isn't the nicest language
  - Especially if you have to write lots of code
  - Buts its getting better (ES6 Modules, templated strings)
- JavaScript isn't the most efficient language
  - Today's browser provide efficient implementations
  - Large operations can tie up the browser
- Responses are asynchronous
- Might need to send large amount of data
  - To cover all possible interactions
  - But data can be sent on demand
- Your base code is public; base data is available
- Normal navigation can be difficult
- Interface can be highly interactive, responsive
- Working with assistive devices & internationalization can be tricky

# Actual Applications




- Mixture of server-side and client-s
- Applications are composed of tas
  - Some tasks are done server-side
  - Some tasks are done client-side
- When developing an application
  - Determine the set of tasks (based on specifications)
  - Determine where/how each task will be done

# CDQuery

Find Your CDs

CD Search:

Find Your CDs

Title

Artist

Track

CD#1 title and major artist
* Track 1
* Track 2

CD#2 title and major artist
* Track 1
* Track 2

CD#3 title and major artist
* Track 1
* Track 2

CD#4 title and major artist
* Track 1

Find Your CDs

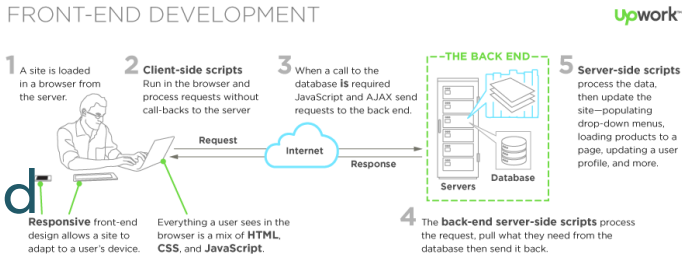
**CD TITLE**
**ARTIST**
**Description**

**TRACK Title**
**Artist**
**Length**
**Description**

**TRACK Title**
**Artist**
**Length**
**Description**

# Client-Side Implementation

- You already have most of the tools needed for this
  - JavaScript to modify the DOM
  - React, Vue, Angular to make this easier
- Client-Side code still needs a back end
  - Data to display has to come from somewhere
  - Results and state need to be stored somewhere
  - Actions need to be taken
- How to communicate with the back end
  - Without replacing the page



# AJAX

- Asynchronous JavaScript And XML
  - JavaScript is used to send an XML request to the server
    - Using a particular URL
    - Expecting XML output as a response
  - When the response comes back, JavaScript runs again
    - Interprets that output
    - Changes the DOM to update the page
- JSON is often used today rather than XML
- JavaScript libraries provide support for this
  - Setting up request; handling response
  - XML, JSON encoding and decoding

# XML

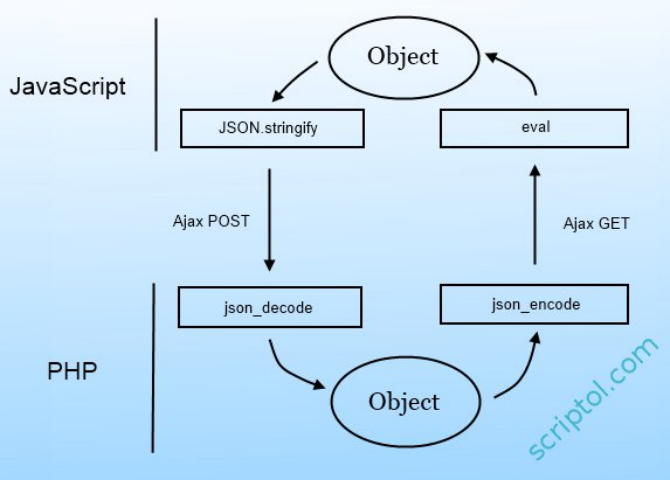
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ARTICLES SYSTEM
"D:\Projects\Clients\XML\Contents\Temp\articlelist.dtd">
<?xml-stylesheet type="text/xsl"
href="D:\xmltohtml.xslt" ?>
<ARTICLES>
    <ARTICLE>
        <ARTICLEDATA>
            <TITLE>XML Demystified</TITLE>
            <AUTHOR>Jaidev</AUTHOR>
        </ARTICLEDATA>
    </ARTICLE>
    <ARTICLE>
        <ARTICLEDATA>
            <TITLE>XSLT Demystified</TITLE>
            <AUTHOR>X S Cel Tea </AUTHOR>
        </ARTICLEDATA>
    </ARTICLE>
    <ARTICLE>
        <ARTICLEDATA>
            <TITLE>C# Demystified</TITLE>
            <AUTHOR>Aleksey N</AUTHOR>
        </ARTICLEDATA>
    </ARTICLE>
</ARTICLES>
```

```
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

XML

# JSON

```
{
    "Rail Booking": {
        "reservation": {
            "ref_no": 1234567,
            "time_stamp": "2016-06-24T14:26:59.125",
            "confirmed": true
        },
        "train": {
            "date": "07/04/2016",
            "time": "09:30",
            "from": "New York",
            "to": "Chicago",
            "seat": "57B"
        },
        "passenger": {
            "name": "John Smith"
        },
        "price": 1234.25,
        "comments": ["Lunch & dinner incl.", "\"Have a nice day!\""]
    }
}
```



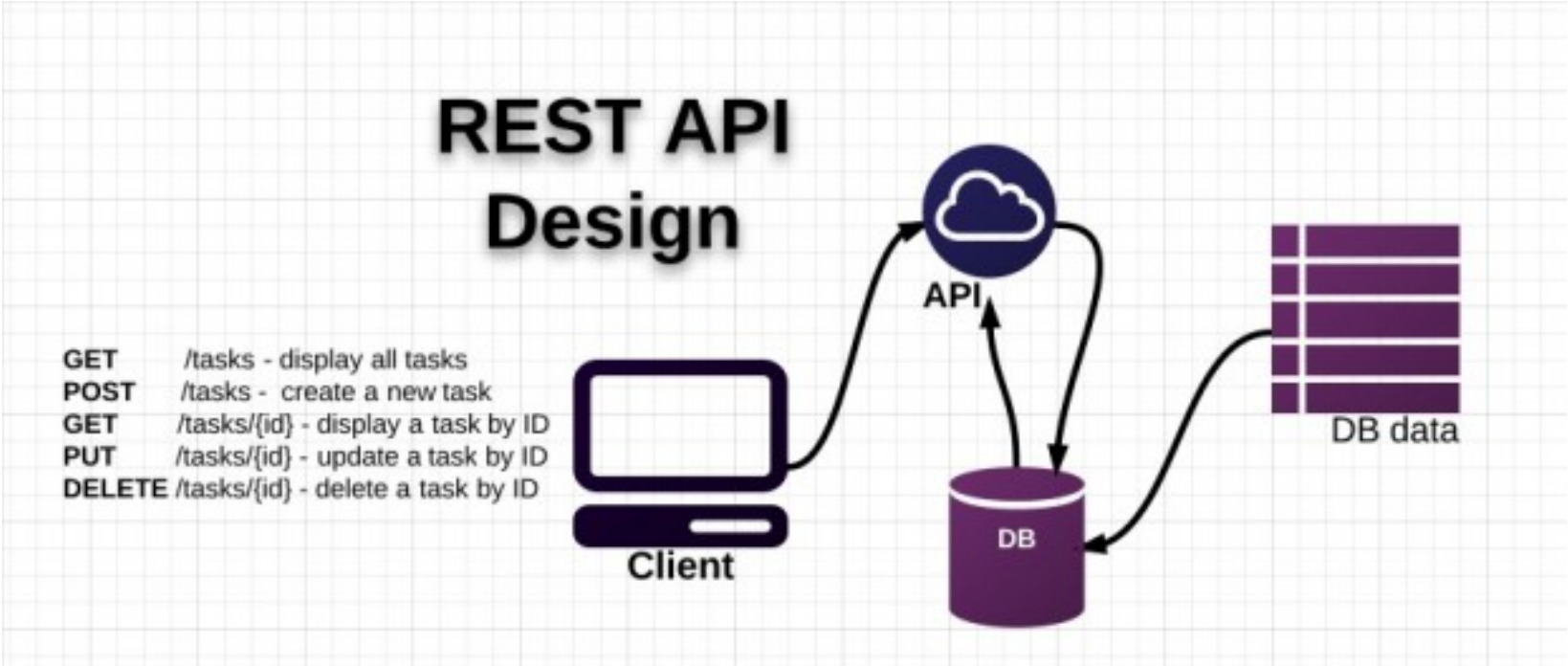
# XMLHttpRequest (Using EcmaScript 6)

- Syntax

```
data = { name: "a name", email: an@email.com };
let p0 = fetch( https://mysite.com/api/query, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(data) } )
    .then( (response) => response.json())
    .then( (data) => { handleData(data); } );
    .catch( (error) => { handleError(data); } );
```

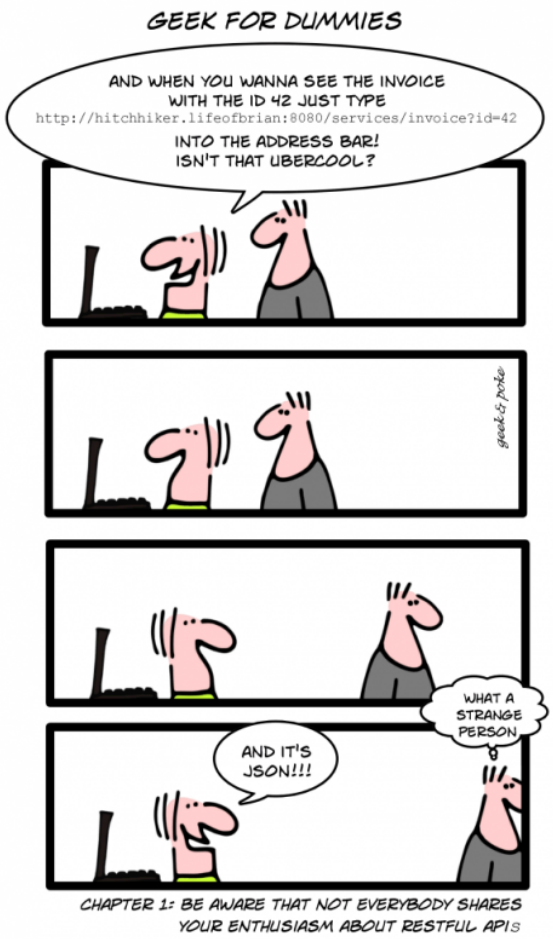
- Request gets sent when JavaScript returns
  - Fetch returns a promise
- Other parameters and events are available
- jQuery has a $.ajax(…) method that is similar

# RESTful APIs

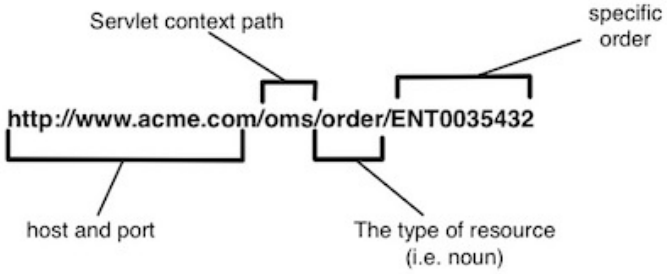


# RESTful APIs

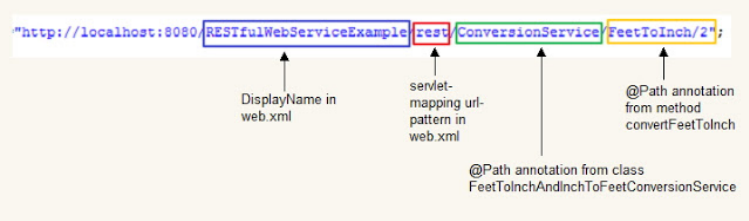
- Use HTTP methods explicitly
  - POST, GET, PUT, DELETE, …
- Are stateless
  - Each request includes all the necessary information
- Expose directory structure-like URLs
  - Use the URL to encode the operation and the data
- Transfer XML or JSON



# URL Encodings

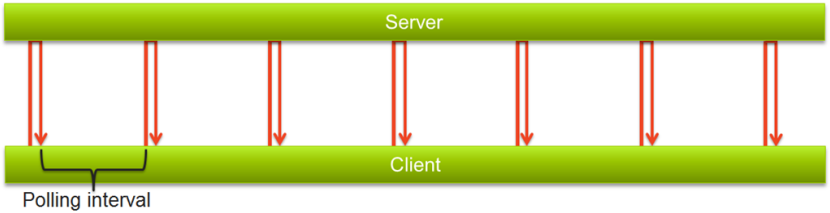



- Suppose we create a chat application
  - POST /chats  with { text: "…", user: "…", title: "…" } => id 01
  - GET /chats/01
  - PUT /chats/01 with { text: "…", user: " …" }
  - DELETE /chats/01
- Can also encode commands
  - GET /command/subcommand/…
  - POST /chats/01/delete
- Can have nested ids
  - GET /command/id/what/id/…

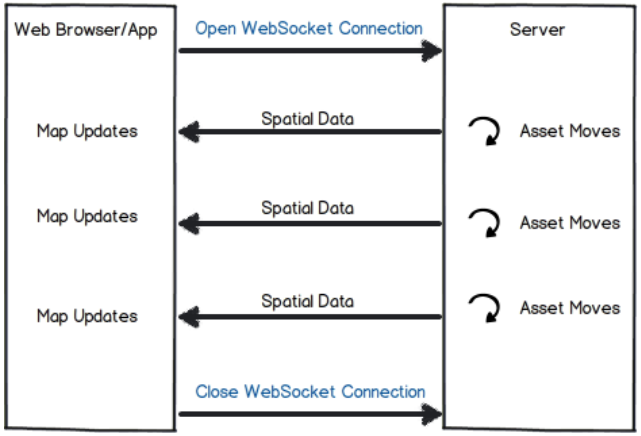


# Front End vs Back End Control

- AJAX implies the front end pulls data from the back end
  - Or posts data as convenient
- What if the back end should be in control
  - Notifications when something unusual happens
  - Continuous information feeds
- One way of handling this is POLLING
  - Front end continually asks the back end "Is anything happening"
  - Not particularly efficient
- There is a better way



# Web Sockets




- AJAX model is client-initiated (pull model)
- Some applications are server-initiated
  - Only want notification when things change
- Web sockets allow this approach
  - Establish a 2-way connection between client and server
  - Send messages from client to server or server to client
  - Messages result in events that trigger code execution
- Handling messages
  - On-events in the client
  - Node.JS events in the server (Socket.IO)
  - Similar support for PHP, Servlets, …

# Socket.IO Server Code

```
var socket = require('socket.io')

function start() {
    … app.get(…) …
    var server = app.listen(port);
    var sio = socket.listen(server);
    sio.socket.on('connection',
    socketConnect);
}
```

```
function socketConnect(s) {
    s.on('usercmd1',
        function(data) { uc1(s,data); });
    s.on('usercmd2',
        function(data) { …} );
    s.on('disconnect',
        function(socket) { … } );
}
function uc1(s,data) {
    s.emit('cmd',{ result: 'xxx' });
}
```

# Socket.IO Client Code

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost');
  socket.on('news', function (data) {
        console.log(data);
        socket.emit('my other event', { my: 'data' }); });
 </script>
```