CSC 470 – Section 3

Topics in Computer Science: Advanced Browser Technologies

Mark F. Russo, Ph.D.

Spring 2016

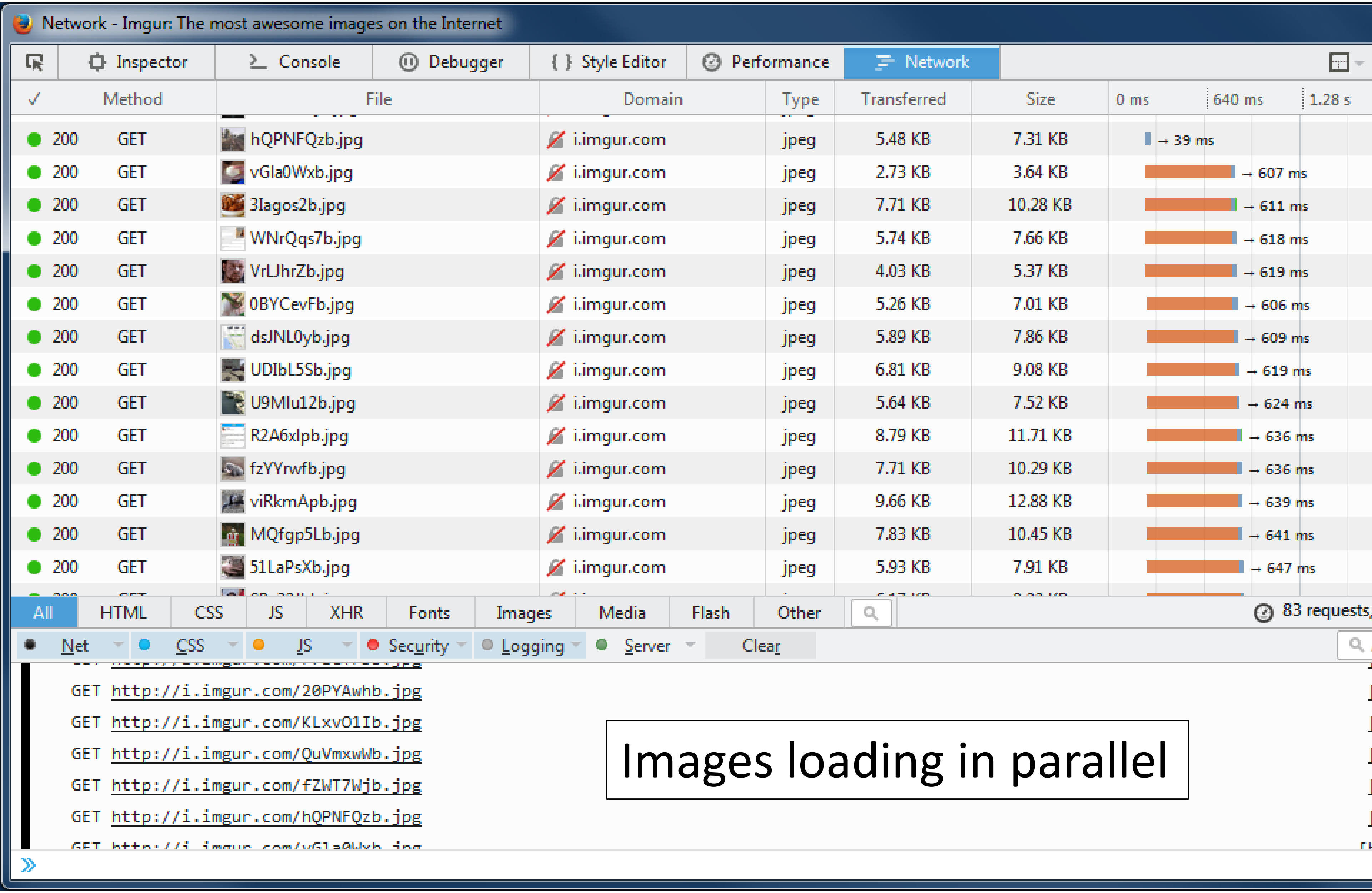
Lecture 10: Asynchronous Programming in JavaScript

Concurrency vs. Parallelism

- A program can be truly parallel only when you have two or more cores or processors that are executing at the same time.
- A concurrent program means that two or more tasks are executing in the same period of time.
- Modern operating systems model concurrency using threads.
  - A thread can be thought of as a single list of tasks to be performed
  - It is the job of the operating system to schedule and switch between threads to create concurrent execution

Asynchronous Behavior

- Browsers perform many actions asynchronously
- Examples:
    - Page resources usually load asynchronously
    - Browser dialogs and JavaScript
    - Browser dialogs and window resizing

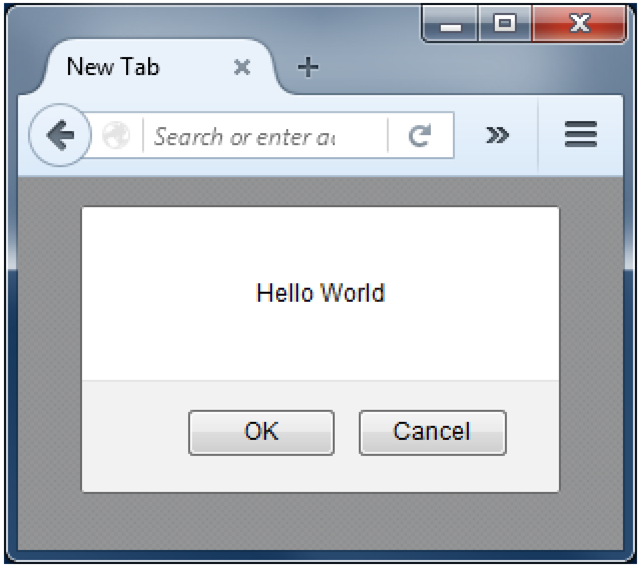


✓		Method	File	Domain	Type	Transferred	Size	0 ms 640 ms 1.28 s
●	200	GET	hQPNFQzb.jpg	i.imgur.com	jpeg	5.48 KB	7.31 KB	→ 39 ms
●	200	GET	vGla0Wxb.jpg	i.imgur.com	jpeg	2.73 KB	3.64 KB	→ 607 ms
●	200	GET	3Iagos2b.jpg	i.imgur.com	jpeg	7.71 KB	10.28 KB	→ 611 ms
●	200	GET	WNrQqs7b.jpg	i.imgur.com	jpeg	5.74 KB	7.66 KB	→ 618 ms
●	200	GET	VrLJhrZb.jpg	i.imgur.com	jpeg	4.03 KB	5.37 KB	→ 619 ms
●	200	GET	0BYCevFb.jpg	i.imgur.com	jpeg	5.26 KB	7.01 KB	→ 606 ms
●	200	GET	dsJNL0yb.jpg	i.imgur.com	jpeg	5.89 KB	7.86 KB	→ 609 ms
●	200	GET	UDIbL5Sb.jpg	i.imgur.com	jpeg	6.81 KB	9.08 KB	→ 619 ms
●	200	GET	U9MIu12b.jpg	i.imgur.com	jpeg	5.64 KB	7.52 KB	→ 624 ms
●	200	GET	R2A6xlpb.jpg	i.imgur.com	jpeg	8.79 KB	11.71 KB	→ 636 ms
●	200	GET	fzYYrwfb.jpg	i.imgur.com	jpeg	7.71 KB	10.29 KB	→ 636 ms
●	200	GET	viRkmApb.jpg	i.imgur.com	jpeg	9.66 KB	12.88 KB	→ 639 ms
●	200	GET	MQfgp5Lb.jpg	i.imgur.com	jpeg	7.83 KB	10.45 KB	→ 641 ms
●	200	GET	51LaPsXb.jpg	i.imgur.com	jpeg	5.93 KB	7.91 KB	→ 647 ms

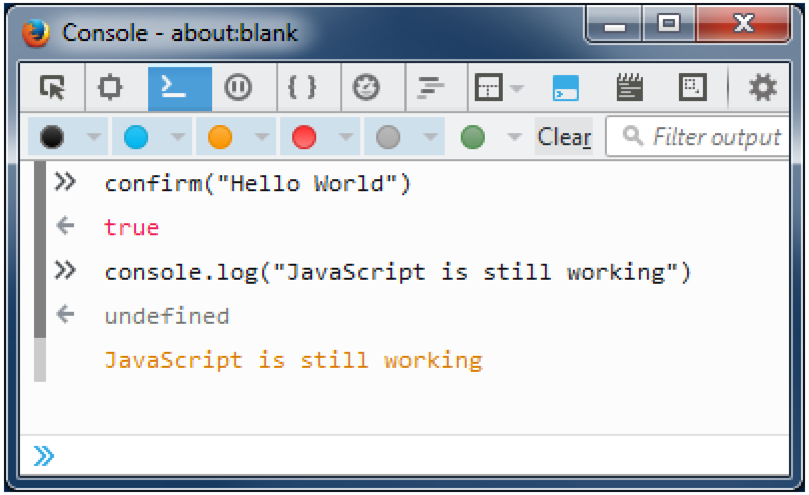
```
GET http://i.imgur.com/20PYAwhb.jpg
GET http://i.imgur.com/KLxvO1Ib.jpg
GET http://i.imgur.com/QuVmxwWb.jpg
GET http://i.imgur.com/fZWT7Wjb.jpg
GET http://i.imgur.com/hQPNFQzb.jpg
GET http://i.imgur.com/vGla0Wxb.jpg
```

Images loading in parallel

Browser Dialogs and JavaScript



Hello World



```
>> confirm("Hello World")
<- true
>> console.log("JavaScript is still working")
<- undefined
   JavaScript is still working
```

- JavaScript continues to respond even though the dialog is displayed and the `confirm(…)` function is paused

Single Threaded Execution

- For the most part, JavaScript in a browser is single threaded
    - Only one thing can happen at a time
    - There are exceptions (alert + browser resize)

- JavaScript is **single threaded on the event loop**
    - Without doing something special, JavaScript executes only one event handler at a time

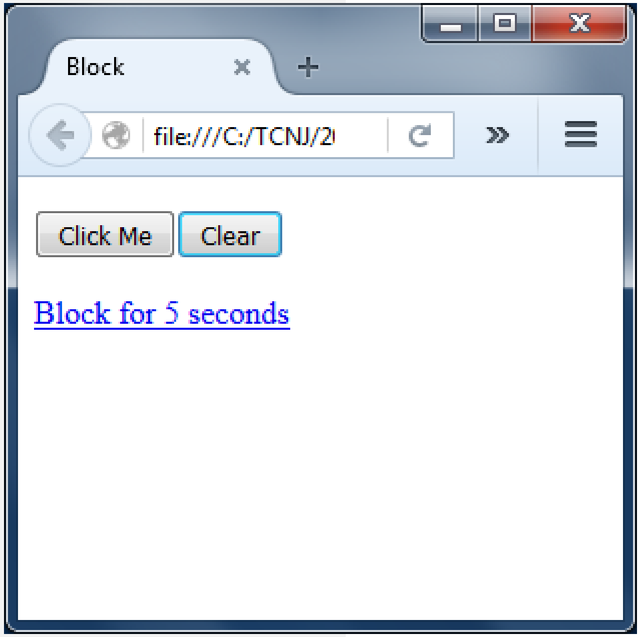
- JavaScript has **run-to-completion semantics**
    - The current task is always finished before the next task begins
    - Each task has complete control over all current state
    - One does not need to worry about concurrent modifications interfering

- Implications
    - While a page DOM is being rendered (on a refresh tick every 16 ms), nothing else can happen
    - When an event fires and triggers a script, nothing else can happen, not even page refreshes
    - If a script gets stuck in a loop, the browser can hang
    - …

- There are ways to address this limitation using concurrency in JavaScript

Blocking the Event Loop

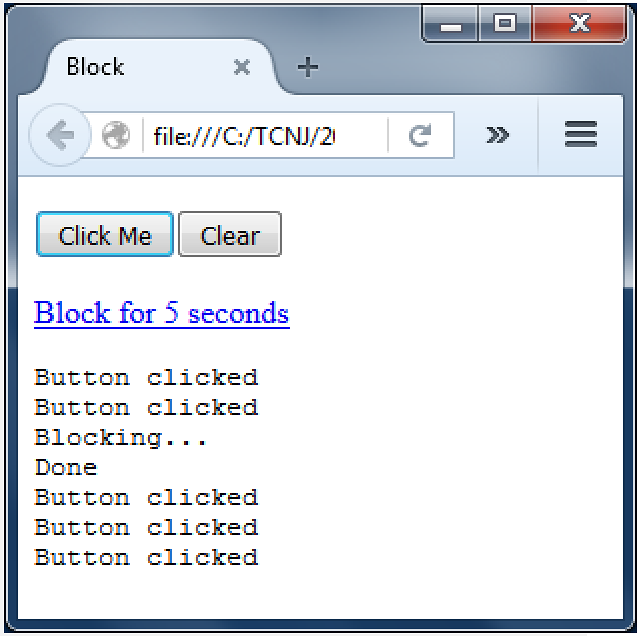
```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Block</title>
  </head>
  <body>
    <p>
      <button id="btnClick">Click Me</button>
      <button id="btnClear">Clear</button></p>
    <p>
      <a id="aBlock" href="">
        Block for 5 seconds
      </a>
    </p>
    <pre id="msg"></pre>

    <script type="text/javascript" src="block.js"></script>
  </body>
</html>
```



http://exploringjs.com/es6/ch_async.html

10/block.html

```
// block.js
(function() {
  // --- Event handlers
  var onAnchorClick = function(ev) {
      ev.preventDefault();

      addMsg('Blocking...');

      // Allows browser to display status message
      setTimeout(function () {
          onTimeout(5000);
          addMsg('Done');
      }, 100);
  };

  var onClick = function(ev) {
    addMsg('Button clicked');
  };

  var onClear = function(ev) {
    document.getElementById('msg').innerHTML = "";
  };

  // --- Functions
  var addMsg = function(msg) {
      var pre = document.getElementById('msg');
      pre.insertAdjacentHTML('beforeend', msg + '\n');
  }
  var onTimeout = function(milliseconds) {
      var start = Date.now();
      while ((Date.now() - start) < milliseconds);
  }

  // --- Set up event handlers
  document.getElementById('aBlock'  ).addEventListener('click', onAnchorClick);
  document.getElementById('btnClick').addEventListener('click', onClick);
  document.getElementById('btnClear').addEventListener('click', onClear);
})();
```



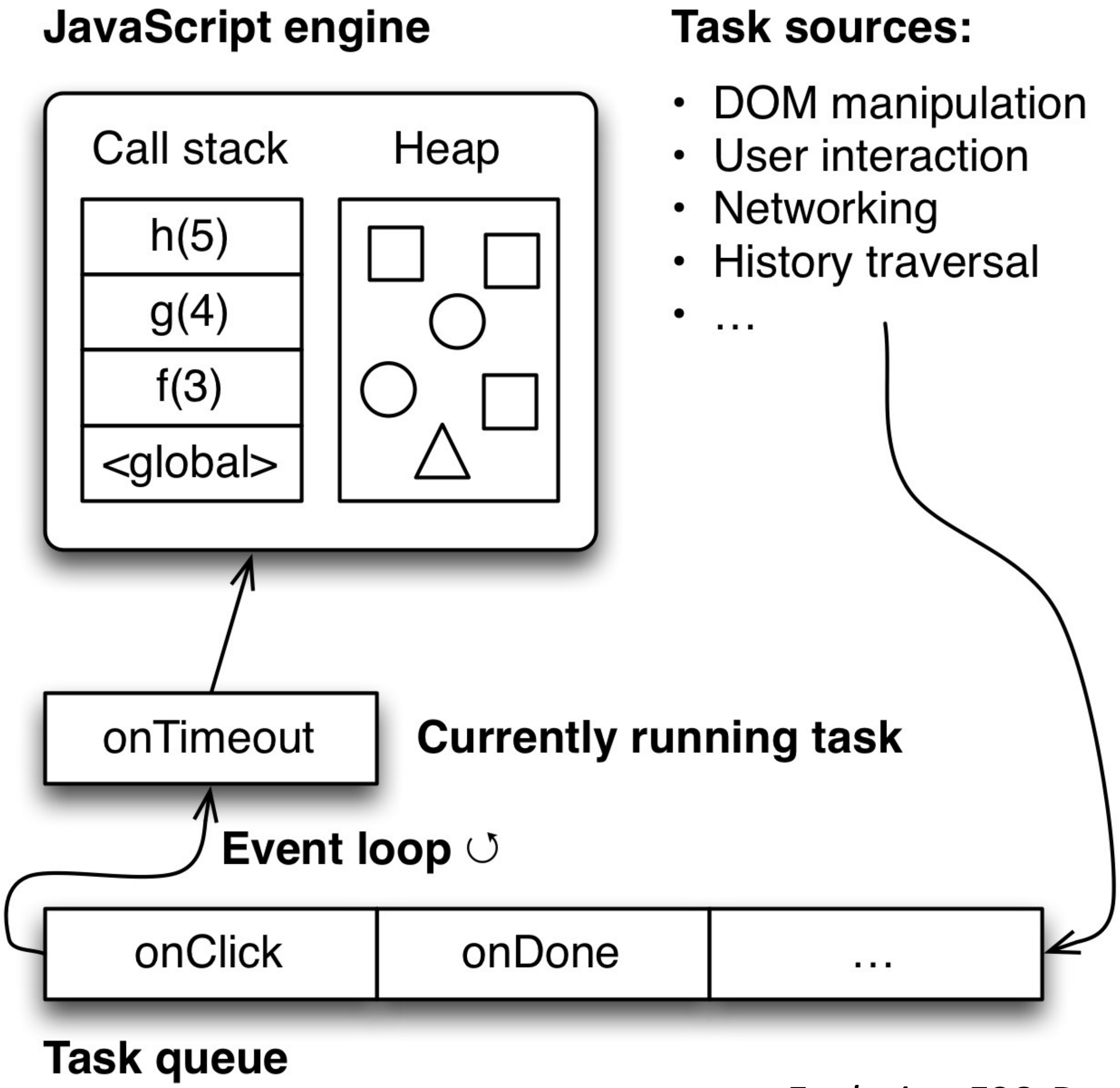
```
Button clicked
Button clicked
Blocking...
Done
Button clicked
Button clicked
Button clicked
```

- Sleeps for 5 seconds on anchor click
- No button events until synchronous sleep completes
- Browser has a single event loop

http://exploringjs.com/es6/ch_async.html

10/block.js

Browser Event Loop



http://exploringjs.com/es6/ch_async.html

Exploring ES6, Dr. Axel Rauschmayer

Asynchronous Execution

Two styles of receiving results asynchronously in JavaScript

1. **Events**
   - Create an object
   - Register event handler functions
   - Handler functions are called on event occurrence
2. **Callbacks**
   - Callback function passed as argument to another asynchronous function
   - When the asynchronous function completes, the callback function is invoked

http://exploringjs.com/es6/ch_async.html

Asynchronous Events: AJAX

- `AJAX`: Asynchronous JavaScript and XML
  - Has nothing to do with XML
- Epitomized by the JavaScript `XMLHttpRequest` object (aka `XHR`)
  - Also, has nothing to do with XML
- The `XHR` object **allows for dynamic asynchronous loading of web resources independent of the standard read/render process**
  - Format of data loaded is not limited to XML
- Term coined in 2005 by Jesse James Garrett

- Use Cases
  - Load HTML fragments and update page dynamically
  - Load data, never intended for rendering
  - …

http://exploringjs.com/es6/ch_async.html

Quick/Simple HTTP Server

Open terminal and enter:

```
python -m SimpleHttpServer    # Python 2
python -m http.server         # Python 3
```

- Files served relative to current directory
- Access using localhost address on port 8000 (by default)

```
http://localhost:8000/
```

Same Origin Policy

- Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but **only if both web pages have the same origin**.

- An origin is defined as a combination of:
  - URI scheme (e.g. http, https, file, …)
  - hostname, and (e.g. tcnj.edu, localhost, …)
  - port number (e.g. 80, 8000, 12345, …)

- This policy **prevents a malicious script** on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

https://en.wikipedia.org/wiki/Same-origin_policy

Using the XHR Object

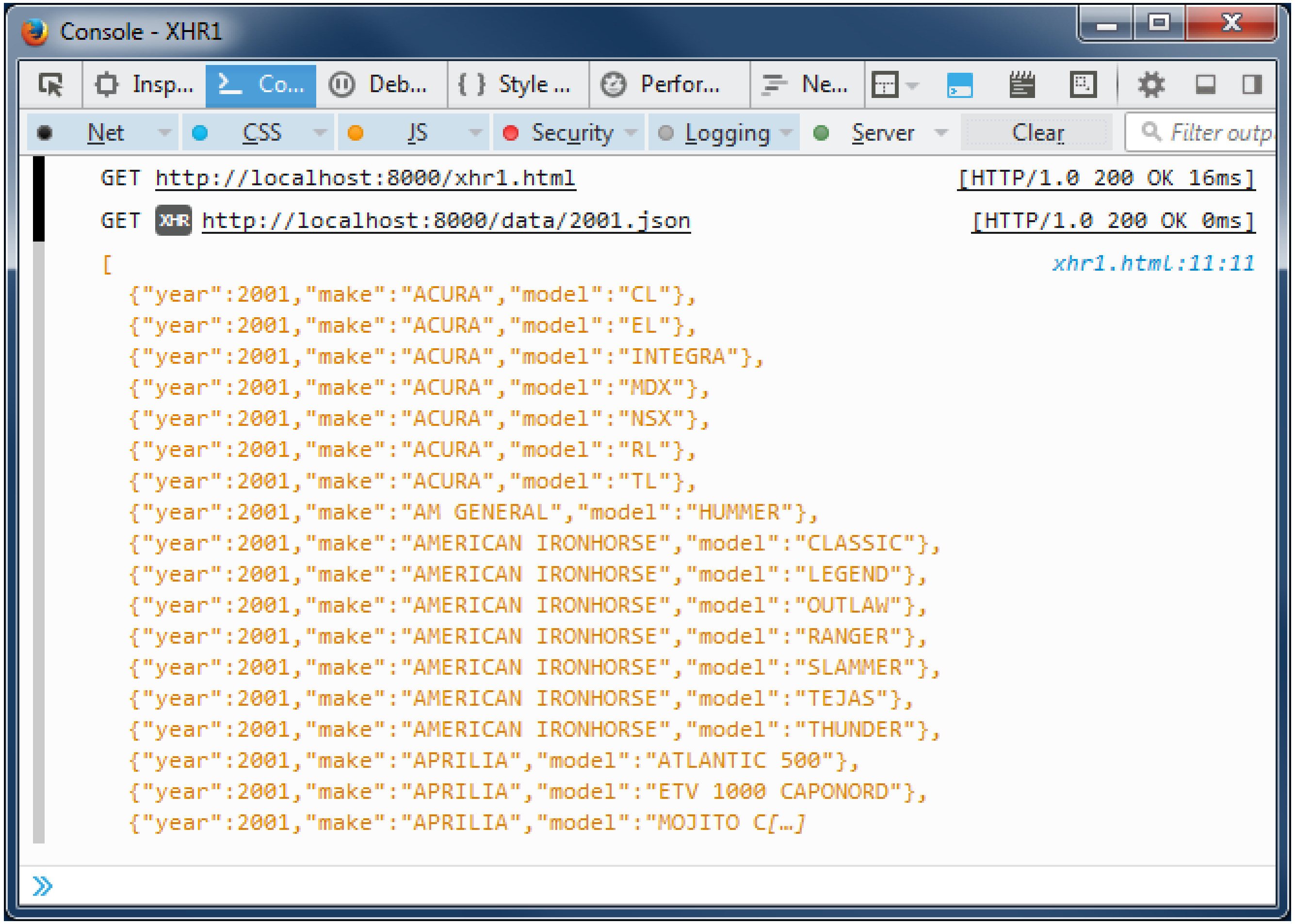
1. Create an instance of an XHR object
2. Set up event listeners
3. Invoke the XHR open(…) method
4. Invoke the XHR send(…) method

```
(function() {

  var showData = function() {
    console.log( this.responseText );
  };

  var xhr = new XMLHttpRequest();       // 1
  xhr.onload = showData;                // 2
  xhr.open("GET", "data/2001.json");    // 3
  xhr.send();                           // 4

})();
```

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest

10/xhr1.html

Using the XHR Object



```
GET http://localhost:8000/xhr1.html                          [HTTP/1.0 200 OK 16ms]
GET XHR http://localhost:8000/data/2001.json                   [HTTP/1.0 200 OK 0ms]
[                                                                    xhr1.html:11:11
  {"year":2001,"make":"ACURA","model":"CL"},
  {"year":2001,"make":"ACURA","model":"EL"},
  {"year":2001,"make":"ACURA","model":"INTEGRA"},
  {"year":2001,"make":"ACURA","model":"MDX"},
  {"year":2001,"make":"ACURA","model":"NSX"},
  {"year":2001,"make":"ACURA","model":"RL"},
  {"year":2001,"make":"ACURA","model":"TL"},
  {"year":2001,"make":"AM GENERAL","model":"HUMMER"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"CLASSIC"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"LEGEND"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"OUTLAW"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"RANGER"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"SLAMMER"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"TEJAS"},
  {"year":2001,"make":"AMERICAN IRONHORSE","model":"THUNDER"},
  {"year":2001,"make":"APRILIA","model":"ATLANTIC 500"},
  {"year":2001,"make":"APRILIA","model":"ETV 1000 CAPONORD"},
  {"year":2001,"make":"APRILIA","model":"MOJITO C[…]
```

XHR API

`open(method, URL, async)`

- Initializes request
- `method`: `String`: HTTP method, such as '`GET`', '`POST`', …
- `URL`: `String`: the address of the resource to load
- `async`: `Boolean`: whether or not the request will be asynchronous

`send([data])`

- Sends the resource request
- `data`: optional data to send as part of request

`setRequestheader(header, value)`

- `header`: name of the HTTP header to set
- `value`: value of the HTTP header

`abort()`

- Aborts the request if the readyState of the XHR object has not yet become 4
- Ensures that the callback handler does not get invoked

`responseText`

- Response to the request as text, or null if the request was unsuccessful or not sent

`response`

- Returns an ArrayBuffer, Blob, Document, JavaScript object, or a DOMString, depending of the value of XMLHttpRequest.responseType

`responseType`

- `"json" | "text" | "blob" | "arraybuffer" | "document"`

`status`

- Integer HTTP status code

`statusText`

- String description of HTTP return status

`readyState`

- Predefined integer identifying state or request

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

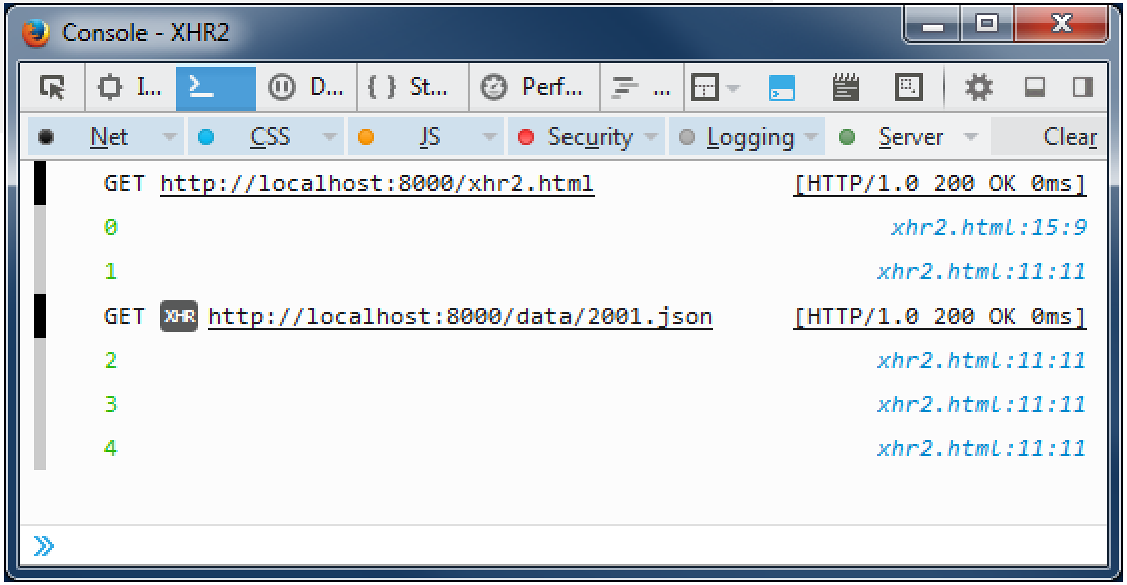
XHR `readystatechange` Event

Property	Description
`onreadystatechange`	A function to be called automatically each time the `readyState` property changes
`readyState`	Holds the status of the `XMLHttpRequest` Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
`status` `statusText`	200: "OK" 404: Page not found

XHR `readystatechange` Example

```
(function() {
  var showState = function() {
    console.log( this.readyState );
  };

  var xhr = new XMLHttpRequest();
  console.log( xhr.readyState );
  xhr.onreadystatechange = showState;
  xhr.open("GET", "data/2001.json");
  xhr.send();
})();
```

asynchronous behavior



```
GET http://localhost:8000/xhr2.html              [HTTP/1.0 200 OK 0ms]
0                                                       xhr2.html:15:9
1                                                      xhr2.html:11:11
GET XHR http://localhost:8000/data/2001.json     [HTTP/1.0 200 OK 0ms]
2                                                      xhr2.html:11:11
3                                                      xhr2.html:11:11
4                                                      xhr2.html:11:11
```

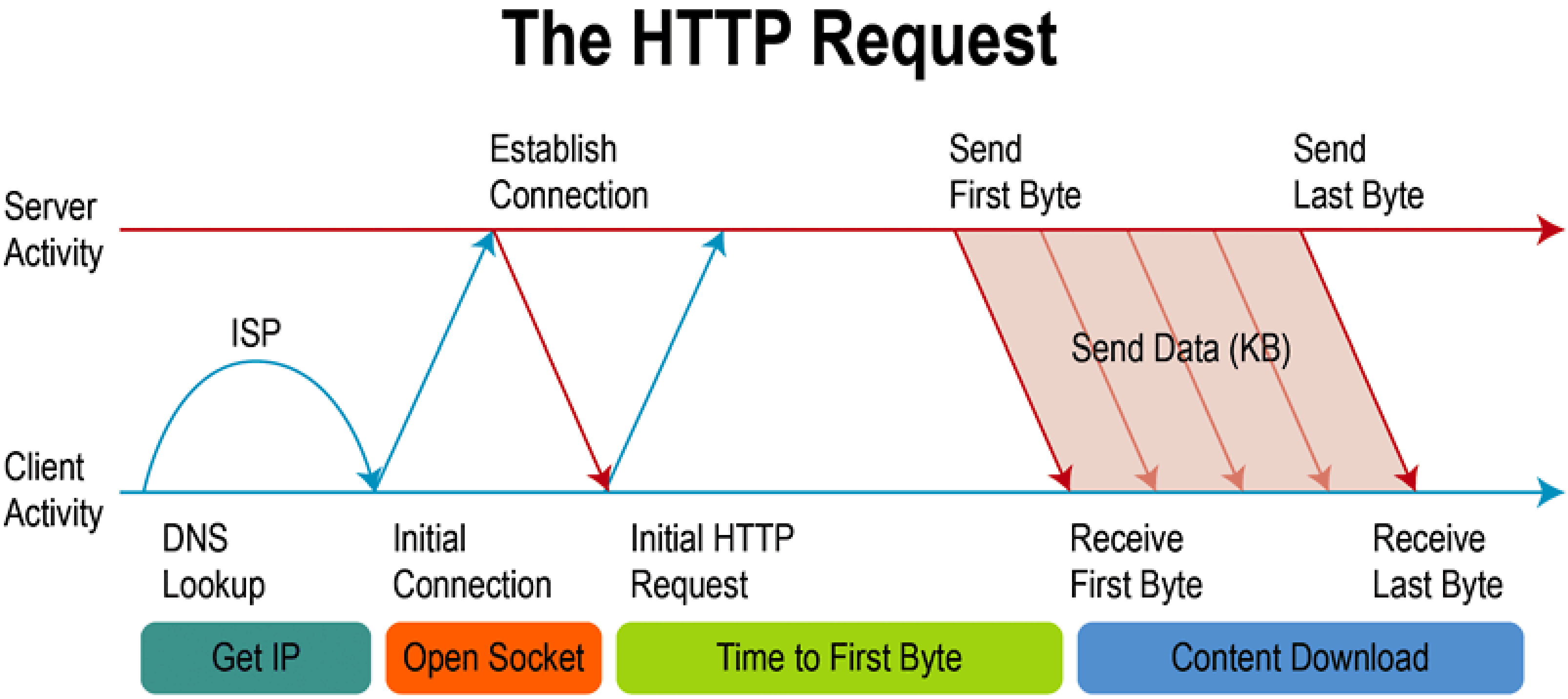
XHR Progress Events

attribute value	Description	Times	When
`loadstart`	Progress has begun.	Once.	First.
`progress`	In progress.	Zero or more.	After `loadstart` has been dispatched.
`error`	Progression failed.	Zero or once.	After the last `progress` has been dispatched, or after `loadstart` has been dispatched if `progress` has not been dispatched.
`abort`	Progression is terminated.	Zero or once.	
`load`	Progression is successful.	Zero or once.	
`loadend`	Progress has stopped.	Once.	After one of `error`, `abort`, or `load` has been dispatched.

https://www.w3.org/TR/progress-events/

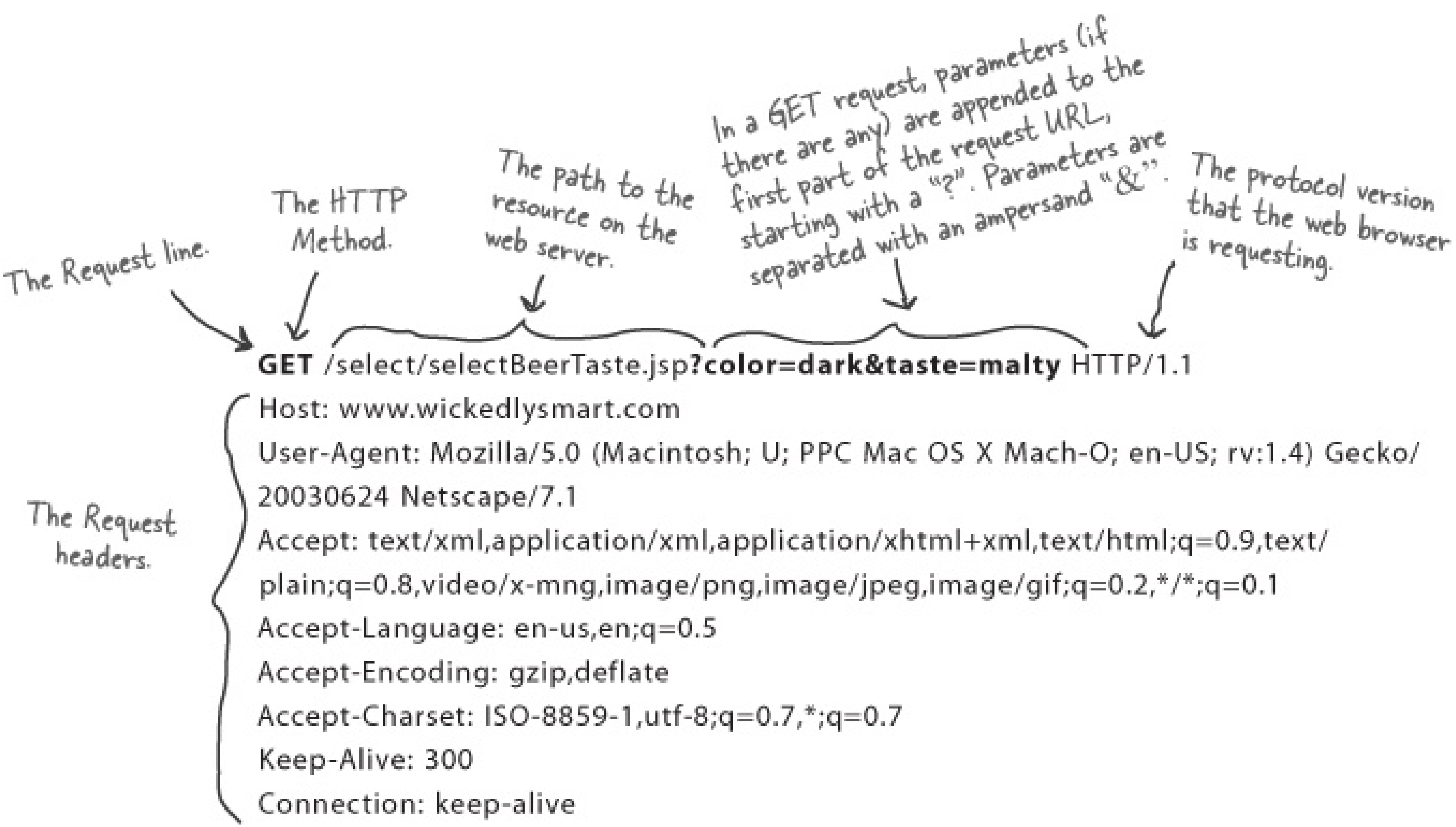
(See 10/xhr1.html)

Anatomy of an HTTP Request



http://www.websiteoptimization.com/secrets/metrics/10-21-http-request.html

Anatomy of an HTTP GET Request



The Request line.

The HTTP Method.

The path to the resource on the web server.

In a GET request, parameters (if there are any) are appended to the first part of the request URL, starting with a "?". Parameters are separated with an ampersand "&".

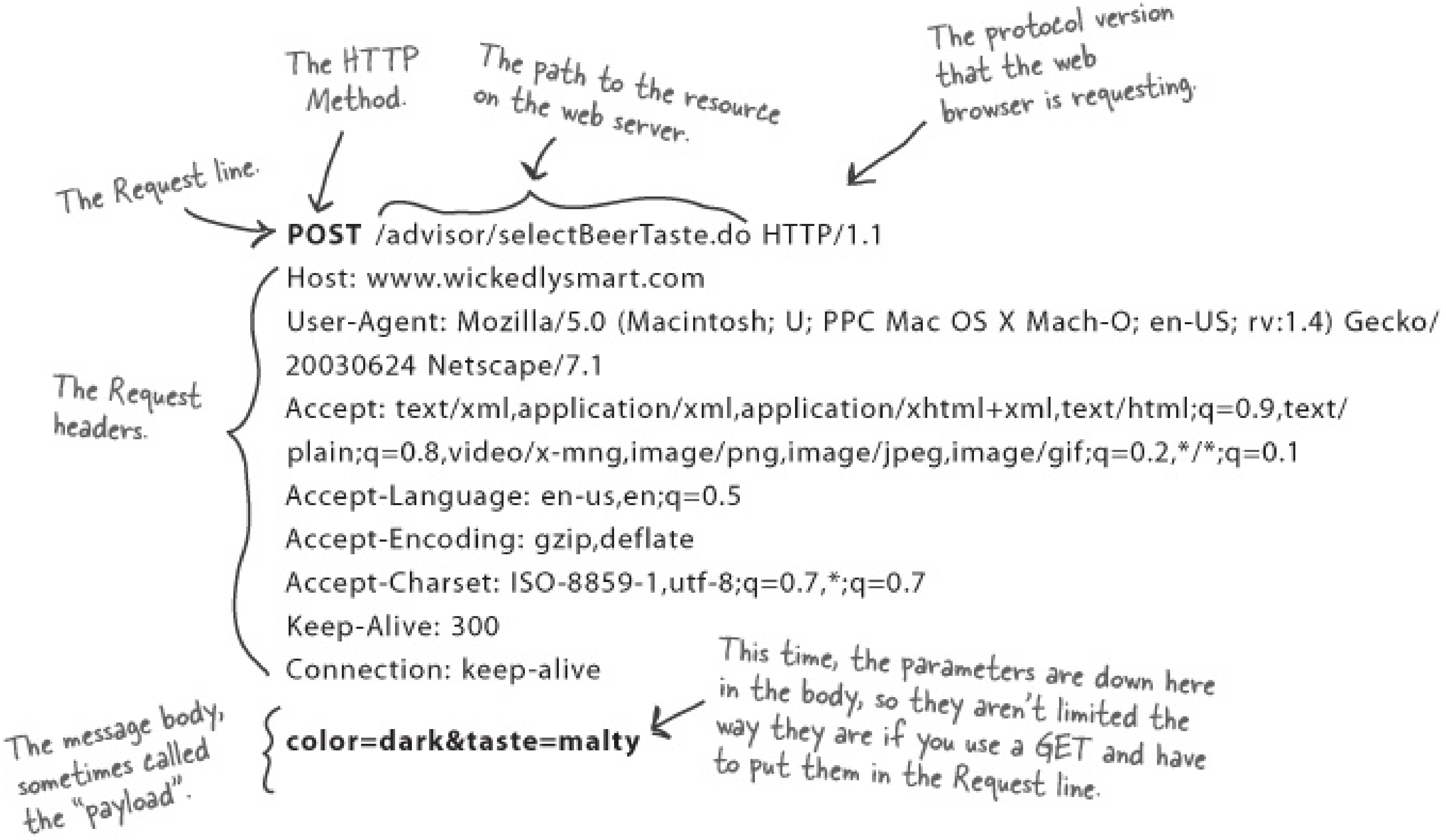
The protocol version that the web browser is requesting.

The Request headers.

```
GET /select/selectBeerTaste.jsp?color=dark&taste=malty HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

https://www.safaribooksonline.com/library/view/head-first-servlets/9780596516680/ch01s13.html

Anatomy of an HTTP POST Request



```
POST /advisor/selectBeerTaste.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

color=dark&taste=malty
```

The HTTP Method.

The path to the resource on the web server.

The protocol version that the web browser is requesting.

The Request line.

The Request headers.

The message body, sometimes called the "payload".

This time, the parameters are down here in the body, so they aren't limited the way they are if you use a GET and have to put them in the Request line.

https://www.safaribooksonline.com/library/view/head-first-servlets/9780596516680/ch01s14.html

GET vs. POST HTTP Methods

GET Requests	**POST Requests**
• Have a size limit of ~2KB-8KB, depending upon browser and version	• Have no size limit
• All parameters in query string. Request body is ignored.	• Transmits data in request body (after request header)
• Permitted to be cached by the browser	• Will not be cached

Synchronous vs. Asynchronous XHR

- Synchronous requests block execution of code which creates "freezing" on the screen and an unresponsive user experience
  - Best practice is to avoid synchronous requests

- Determined by the third parameter of the `open()` method (a Boolean)
  - `true`: synchronous request
  - `false`: asynchronous request (default)

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

Asynchronous Execution

Two styles of receiving results asynchronously in JavaScript

1. **Events**
    - Create an object
    - Register event listeners
    - Listener functions are called on event occurrence

2. **Callbacks**
    - Callback function passed as argument to another asynchronous function
    - When complete, callback function is invoked

http://exploringjs.com/es6/ch_async.html

Asynchronous Execution: Recall Timers

JavaScript functions can be scheduled to run after a period of time, on a regular time interval, or prior to the next repaint of the window

```
var id = setTimeout(callback, delay)
```

- Invokes a function or executes a code snippet after specified delay

```
clearTimeout(id)
```

- Clears the delay set by `setTimeout()`

```
var id = setInterval(callback, delay)
```

- Invokes a function or executes a code snippet repeatedly, with a fixed time delay between each call

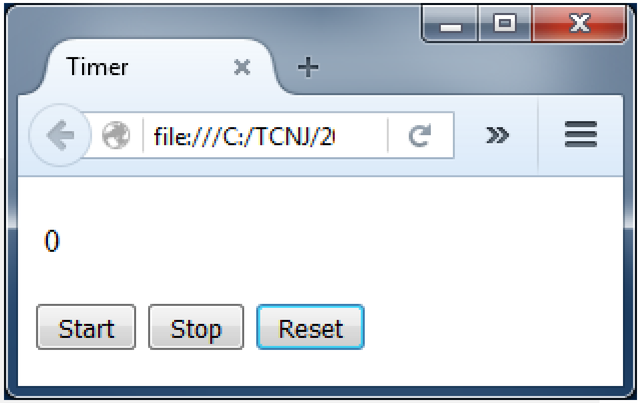
```
clearInterval(id)
```

- Clears the delay set by `setTimeout()`

The timer pauses for `delay` milliseconds and then adds the function `callback` to the task queue.

https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers

Example: Simple Timer



```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Timer</title>
    <script type="text/javascript" src="timer.js">
    </script>
  </head>

  <body>
    <p style="padding: 5px;" id="readout">0</p>
    <p>
      <button onclick="startTimer();">Start</button>
      <button onclick="stopTimer();"> Stop </button>
      <button onclick="resetTimer();">Reset</button>
    </p>
  </body>
</html>
```

05/timer.html

Example: Simple Timer

```javascript
// timer.js

// globals
var tenths = 0.0;
var timerID = null;

// Start the timer running at a 100 millisec interval
var startTimer = function() {
  timerID = setInterval(update, 100);
};

// The function called on each interval
// Add one tenth to counter and update display
var update = function(time) {
  tenths += 1;
  updateDisplay();
};

// Stop the timer using the saved timerID
var stopTimer = function() {
  if (timerID !== null) clearInterval(timerID);
  timerID = null;
}

// Reset the tenths counter and update display
var resetTimer = function() {
  tenths = 0.0;
  updateDisplay();
};

// Reset the timer display
var updateDisplay = function() {
  var p = document.getElementById('readout');
  p.innerHTML = (tenths/10).toString();
}
```

Approximately every 100 milliseconds an instance of update() is added to the task queue

05/timer.js

JavaScript: Run-to-completion Semantics

```
var test = function() {

  setTimeout( function () { console.log('Task 2'); },
              0 );

  console.log('Task 1');
};

test();
```

```
Task 1
Task 2
```

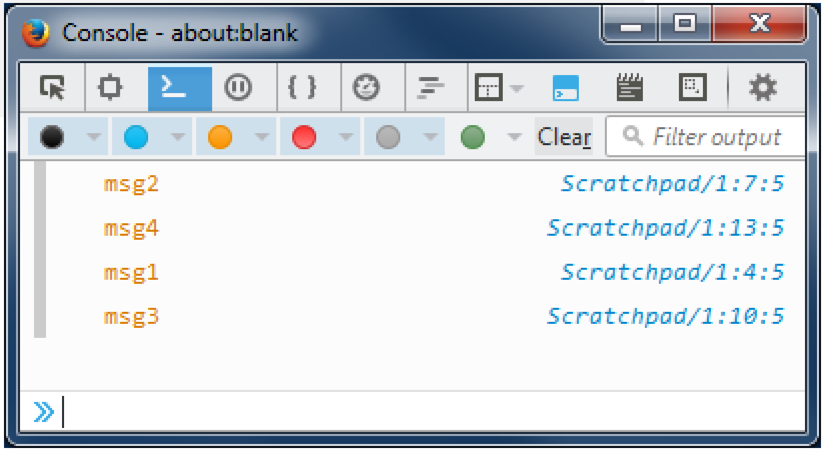
- The anonymous function in `setTimeout()` is added to the task queue immediately.
- It is not executed until after the current function completes.
- The output is deterministic: it will always be the same.

http://exploringjs.com/es6/ch_async.html

Asynchronous Tasks

Asynchronous tasks return when complete, which may not be in the order desired

```
var tasks = function() {

  setTimeout( function() { console.log('msg1'); },
                3000);
  setTimeout( function() { console.log('msg2'); },
                1000);
  setTimeout( function() { console.log('msg3'); },
                4000);
  setTimeout( function() { console.log('msg4'); },
                2000);
}
```



```
msg2                                   Scratchpad/1:7:5
msg4                                   Scratchpad/1:13:5
msg1                                   Scratchpad/1:4:5
msg3                                   Scratchpad/1:10:5
```

10/timer_sequence.js

Example: Asynchronous Sequencing

```
var tasks = function() {
  setTimeout( function() { console.log('msg1'); },
              3000);
};
```

- We want to sequence the functions that print 'msg1', 'msg2', 'msg3', 'msg4'
- To cause asynchronous tasks to execute in order, a **callback function** is passed as an argument and invoked when the current asynchronous task completes

10/timer_sequence.js

Example: Asynchronous Sequencing

```
var tasks = function() {
  setTimeout( function() {
    console.log('msg1');
    setTimeout( function() {
      console.log('msg2');
    }, 1000);
  }, 3000);
};
```

10/timer_sequence.js

Example: Asynchronous Sequencing

```
var tasks = function() {
  setTimeout( function() {
    console.log('msg1');
    setTimeout( function() {
      console.log('msg2');
      setTimeout( function() {
        console.log('msg3')
        }, 4000);
      }, 1000);
  }, 3000);
};
```

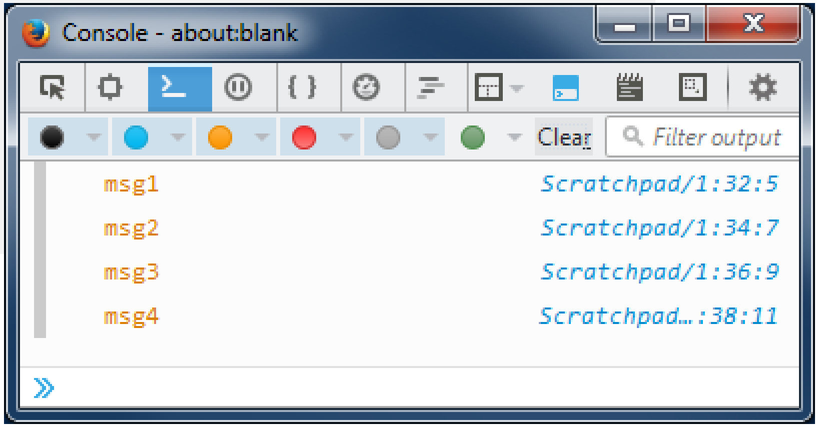
10/timer_sequence.js

Example: Asynchronous Sequencing

```
var tasks = function() {
  setTimeout( function() {
    console.log('msg1');
    setTimeout( function() {
      console.log('msg2');
      setTimeout( function() {
        console.log('msg3');
        setTimeout( function() {
          console.log('msg4');
        }, 2000);
      }, 4000);
    }, 1000);
  }, 3000);
};
```

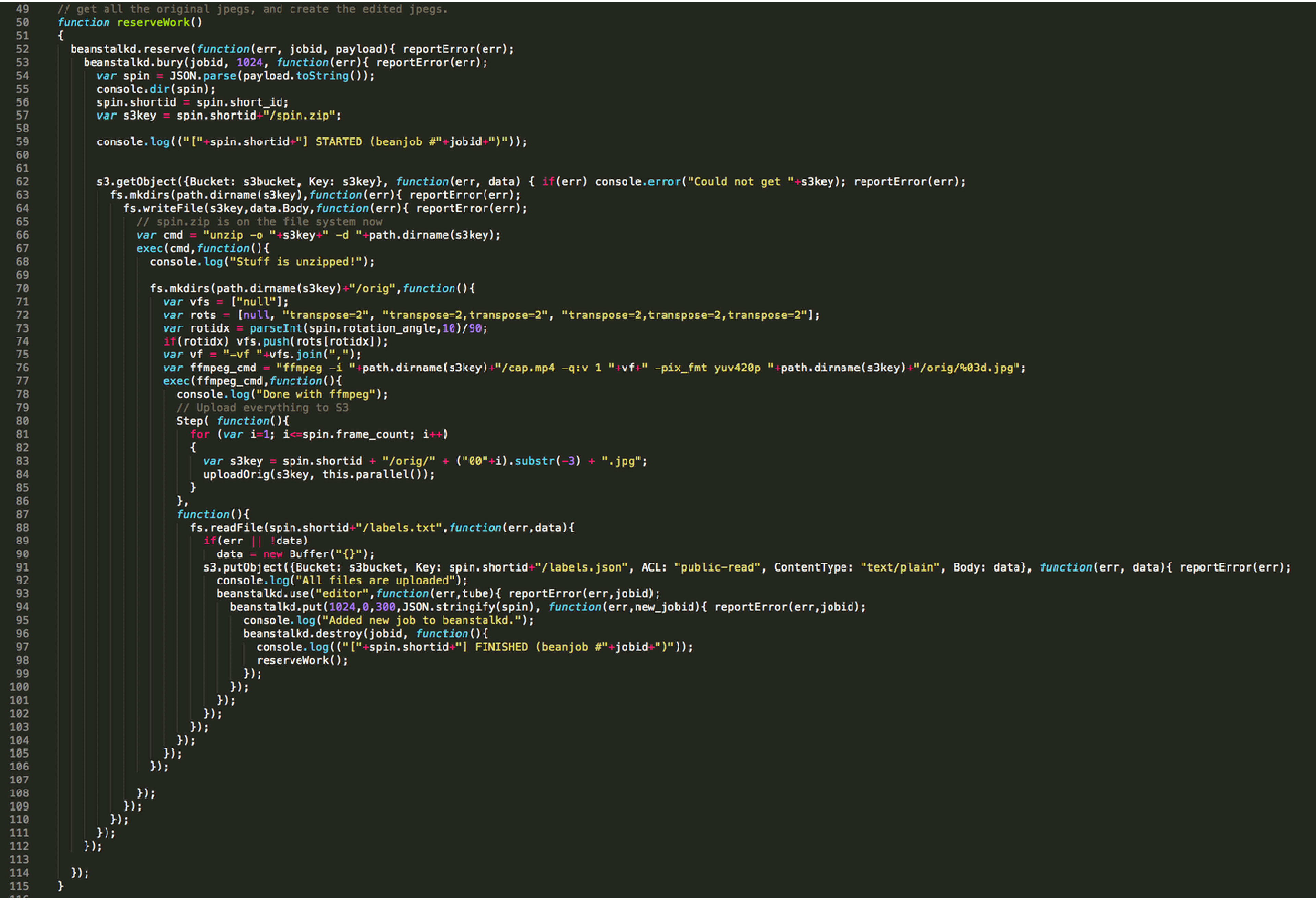
Callback Hell

- Successfully sequences functions
- Can be hard to create and manage



```
msg1                                Scratchpad/1:32:5
msg2                                Scratchpad/1:34:7
msg3                                Scratchpad/1:36:9
msg4                                Scratchpad…:38:11
```

10/timer_sequence.js



```
// get all the original jpegs, and create the edited jpegs.
function reserveWork()
{
  beanstalkd.reserve(function(err, jobid, payload){ reportError(err);
    beanstalkd.bury(jobid, 1024, function(err){ reportError(err);
      var spin = JSON.parse(payload.toString());
      console.dir(spin);
      spin.shortid = spin.short_id;
      var s3key = spin.shortid+"/spin.zip";

      console.log(("["+spin.shortid+"] STARTED (beanjob #"+jobid+")"));


      s3.getObject({Bucket: s3bucket, Key: s3key}, function(err, data) { if(err) console.error("Could not get "+s3key); reportError(err);
        fs.mkdirs(path.dirname(s3key),function(err){ reportError(err);
          fs.writeFile(s3key,data.Body,function(err){ reportError(err);
            // spin.zip is on the file system now
            var cmd = "unzip -o "+s3key+" -d "+path.dirname(s3key);
            exec(cmd,function(){
              console.log("Stuff is unzipped!");

              fs.mkdirs(path.dirname(s3key)+"/orig",function(){
                var vfs = ["null"];
                var rots = [null, "transpose=2", "transpose=2,transpose=2", "transpose=2,transpose=2,transpose=2"];
                var rotidx = parseInt(spin.rotation_angle,10)/90;
                if(rotidx) vfs.push(rots[rotidx]);
                var vf = "-vf "+vfs.join(",");
                var ffmpeg_cmd = "ffmpeg -i "+path.dirname(s3key)+"/cap.mp4 -q:v 1 "+vf+" -pix_fmt yuv420p "+path.dirname(s3key)+"/orig/%03d.jpg";
                exec(ffmpeg_cmd,function(){
                  console.log("Done with ffmpeg");
                  // Upload everything to S3
                  Step( function(){
                    for (var i=1; i<=spin.frame_count; i++)
                    {
                      var s3key = spin.shortid + "/orig/" + ("00"+i).substr(-3) + ".jpg";
                      uploadOrig(s3key, this.parallel());
                    }
                  },
                  function(){
                    fs.readFile(spin.shortid+"/labels.txt",function(err,data){
                      if(err || !data)
                        data = new Buffer("{}");
                      s3.putObject({Bucket: s3bucket, Key: spin.shortid+"/labels.json", ACL: "public-read", ContentType: "text/plain", Body: data}, function(err, data){ reportError(err);
                        console.log("All files are uploaded");
                        beanstalkd.use("editor",function(err,tube){ reportError(err,jobid);
                          beanstalkd.put(1024,0,300,JSON.stringify(spin), function(err,new_jobid){ reportError(err,jobid);
                            console.log("Added new job to beanstalkd.");
                            beanstalkd.destroy(jobid, function(){
                              console.log(("["+spin.shortid+"] FINISHED (beanjob #"+jobid+")"));
                              reserveWork();
                            });
                          });
                        });
                      });
                    });
                  });
                });
              });

            });
          });
        });
      });
    });

  });
}
```

http://slides.com/michaelholroyd/asyncnodejs#/2/2

Continuation-Passing Style (CPS)

- The programming style of using callbacks is also called **continuation-passing style** (CPS)
  - the next step (the continuation) is explicitly passed as a parameter.
- For each step, the control flow of the program continues with the callback function.
- Error handling becomes more complicated
  - Errors are reported via callbacks and via exceptions

http://exploringjs.com/es6/ch_async.html

Promises

- Promises are objects that make working with asynchronous operations much more pleasant.
- Promises help sequence asynchronous functions
  - There is no clean way to wait for an asynchronous function to complete
  - You wouldn't want to wait due to JavaScript's single threaded execution
- Promises provide a better way of working with callbacks
  - Promises help manage callback hell
- As an object, a Promise may be passed like any value

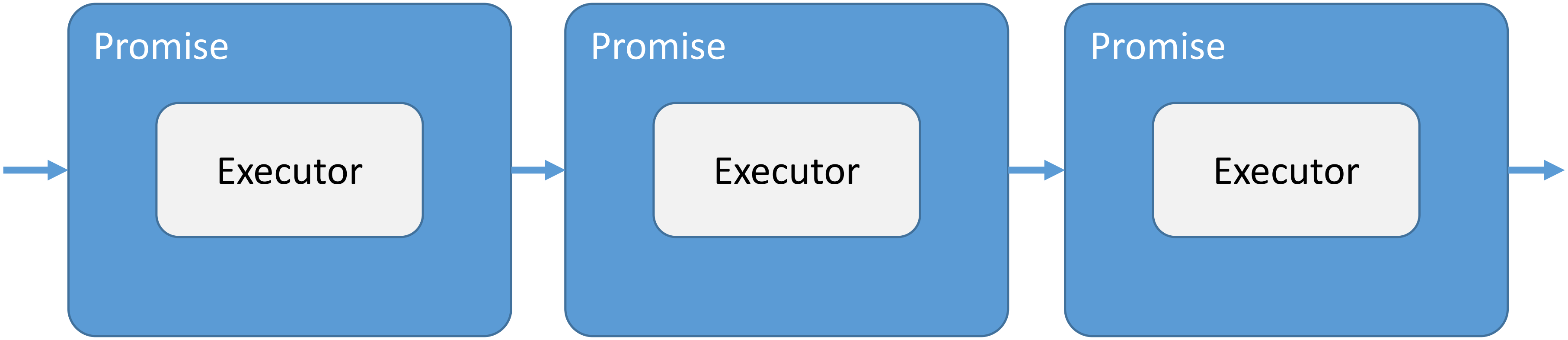
A Promise is a "promise" that a result is forthcoming, and it can act as a proxy for the future result

http://exploringjs.com/es6/ch_promises.html

Promises

Promises wrap and manage asynchronous (or synchronous) functions

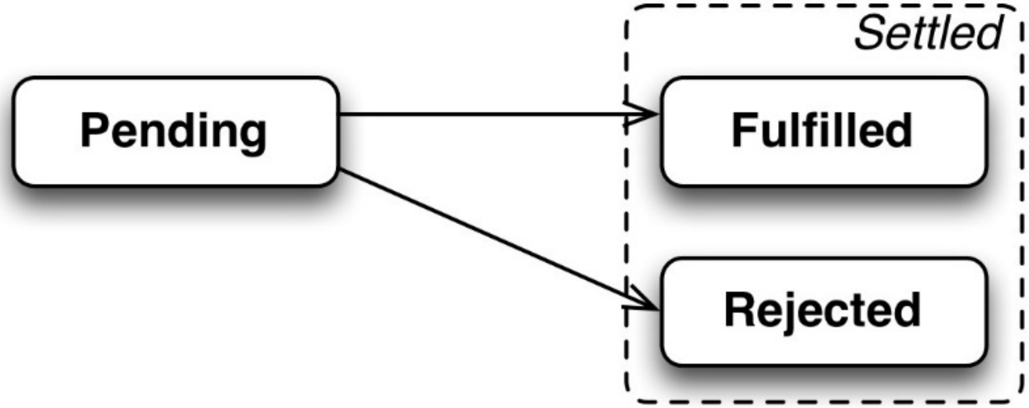
1. Execute Function (called the Executor)
2. Maintain State of Executor (pending, fulfilled, rejected)
3. Can detect and react to successful completion or an error
4. May be chained in sequence in a natural manner ("thenable")



Instead of nesting asynchronous functions in a manner that produces Callback Hell, Promises allow us to chain them and execute them in sequence

Promise States

- A Promise is always in one of three mutually exclusive states:
  - Before the result is ready, the Promise is pending.
  - If a result is available, the Promise is fulfilled.
  - If an error happened, the Promise is rejected.
- A Promise is settled if “things are done” (if it is either fulfilled or rejected).
- A Promise is settled exactly once and then can no longer be changed.



http://exploringjs.com/es6/ch_promises.html

Creating a Promise: Step 1

- The function managed by a Promise (Executor) is passed two arguments:

  1. A function to be invoked when the Executor resolves successfully
  2. A function to be invoked when the Executor has an error of any sort and rejects the outcome

- Example: Wrapping an asynchronous function producing a suitable Executor
  - a.k.a. "Promisifying" a function

```
var doWork = function( resolve, reject ) {

  setTimeout( function() {
    console.log(msg);
    resolve(msg);          // Resolved
    // reject(msg);        // Rejected
  }, delay);

};
```

Creating a Promise: Step 2

- Invoke the Promise object constructor with the Executor as its parameter

```
var doTask = function(msg, delay) {

  var doWork = function( resolve, reject ) {
    setTimeout( function() {
      console.log(msg);
      resolve(msg);         // Resolved
      // reject(msg);       // Rejected
    }, delay);
  };

  return new Promise( doWork );
};
```

Promisified Function

New Promise Object

10/timer_sequence.js

Invoking a Promise

```
var doTask = function(msg, delay) {

  var doWork = function( resolve, reject ) {
    setTimeout( function() {
      console.log(msg);
      resolve(msg);        // Resolved
      // reject(msg);      // Rejected
    }, delay);
  };

  return new Promise( doWork );
};

doTask('msg1', 3000);
```

- A 3 second delay followed by the message.
- So what did we gain?

Responding to Settled Promises

- Promises have `then()` and `catch()` methods.
  - `then()` defines what to do when the promise is <u>resolved</u>
  - `catch()` defines what to do when the promise is <u>rejected</u>

```
var tasks = function() {
  doTask('msg1', 3000)
  .then(  function(msg) { return doTask('msg2', 1000); } )
  .catch( function(reason) { console.log('rejected'); } );
}

tasks();
```

- Alternatively, `then(…)` can take two parameters: `resolved` and `rejected`

```
var tasks = function() {
  doTask('msg1', 3000)
  .then(  function(msg) { return doTask('msg2', 1000); },
          function(reason) { console.log('rejected'); } );
}
```

10/timer_sequence.js

Chaining Promises

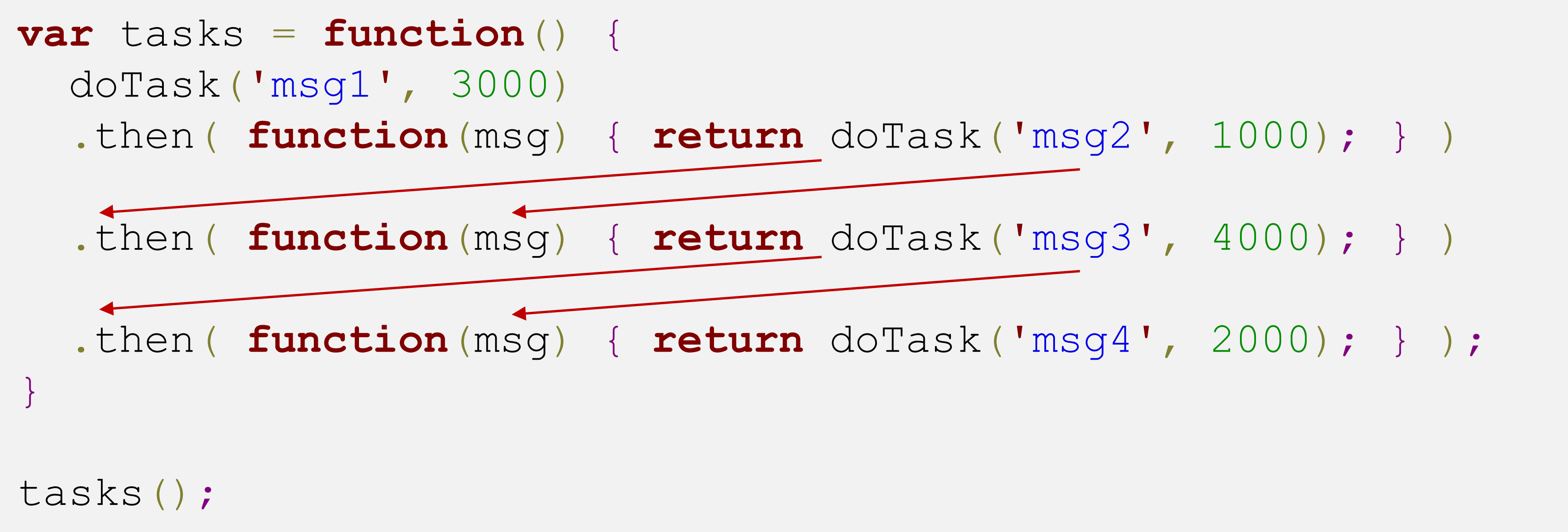
- The `Promise.prototype.then()` and `Promise.prototype.catch()` methods return new Promises
- Promises can be chained - called *composition*
- If a value is used to resolve or reject a Promise, it is automatically wrapped in a new Promise object before being returned. This makes it chainable.

```
var tasks = function() {
  doTask('msg1', 3000)
  .then( function(msg) { return doTask('msg2', 1000); } )
  .then( function(msg) { return doTask('msg3', 4000); } )
  .then( function(msg) { return doTask('msg4', 2000); } );
}

tasks();
```

10/timer_sequence.js

Promises are called "thenable"

- A *thenable* is an object that has a Promise-style `then()` method.
- If the promise has already been fulfilled and later you attach a `then()` to it with callbacks, the success callback will be correctly called.
- We are not interested in knowing when the promise is settled. We are only concerned with the final outcome of the promise.



```
var tasks = function() {
  doTask('msg1', 3000)
  .then( function(msg) { return doTask('msg2', 1000); } )

  .then( function(msg) { return doTask('msg3', 4000); } )

  .then( function(msg) { return doTask('msg4', 2000); } );
}

tasks();
```

http://exploringjs.com/es6/ch_promises.html

Promises and Synchronous Functions

- Promises can be used to manage synchronous functions as well
- Returned values are wrapped automatically into new Promise objects
- Also demonstrates that functions are executed in sequence

```
// Adding synchronous functions
var tasks = function() {
  // asynchronous
  doTask('msg1', 3000)
  // synchronous
  .then( function(msg) { console.log("between 1 and 2"); return msg;} )
  // asynchronous
  .then( function(msg) { return doTask('msg2', 1000); } )
  // synchronous
  .then( function(msg) { console.log("between 2 and 3"); return msg;} )
  // asynchronous
  .then( function(msg) { return doTask('msg3', 4000); } )
  // synchronous
  .then( function(msg) { console.log("between 3 and 4"); return msg;} )
  // asynchronous
  .then( function(msg) { return doTask('msg4', 2000); } )
  // synchronous
  .then( function(msg) { console.log("All done") } );
}

tasks();
```

10/timer_sequence.js

Resolving Multiple Promises

- Promises offer a way to execute multiple Promise executers in parallel, and to wait until all are settled
- The `Promise.all(` *`array`* `)` static function resolves when all Promises in *`array`* resolve
- Results from all Promises are passed as the argument to the next `then()` function
- The first rejection is passed to `catch()`

Resolving Multiple Promises - Example

```
// all.js

// Function to create a promise that wraps a timer
var doTask = function(msg, delay) {

  var doWork = function( resolve, reject ) {
    setTimeout( function() {
      //console.log(msg);
      resolve(msg);
    }, delay);
  };

  return new Promise( doWork );
};

// Create and run all three Promises, simultaneously
Promise.all([doTask('msg1', 3000),
             doTask('msg2', 2000),
             doTask('msg3', 1000)])
.then( function(msgs) { console.log(msgs); } );
```

```
Array [ "msg1", "msg2", "msg3" ]
```

Promise Racing

- Promises offer a way to execute multiple Promise executers in parallel, and to wait until the first is settled
- The `Promise.race( `*`array`*` )` static function resolves when the first Promises in *`array`* resolve
- Result from the first Promise is passed as the argument to the next `then()` function
- The first rejection is passed to `catch()`

Promise Racing - Example

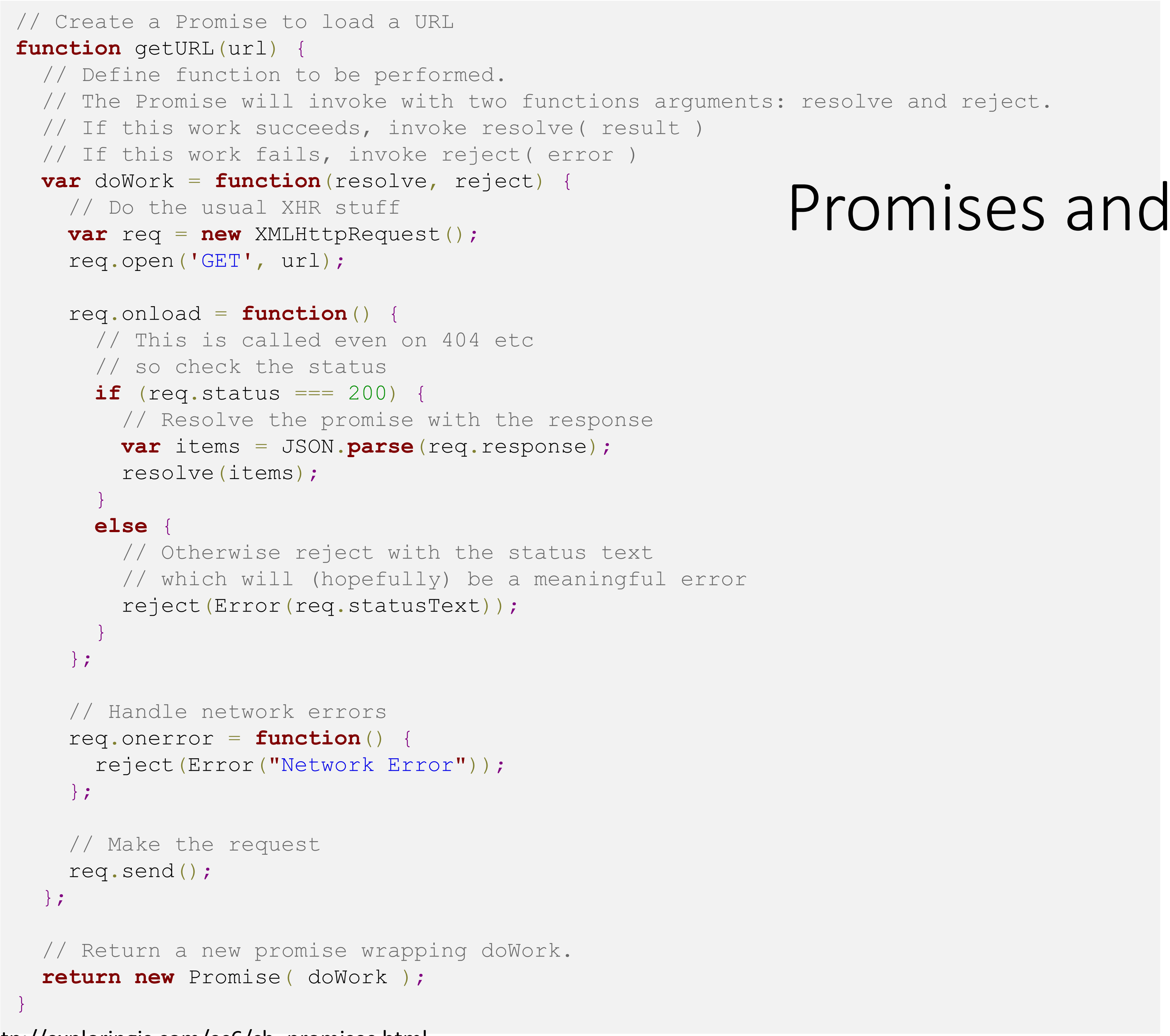
```
// race.js

// Function to create a promise that wraps a timer
var doTask = function(msg, delay) {

  var doWork = function( resolve, reject ) {
    setTimeout( function() {
      //console.log(msg);
      resolve(msg);
    }, delay);
  };

  return new Promise( doWork );
};

// Run a race between all three Promises
Promise.race([doTask('msg1 wins!', 3000),
              doTask('msg2 wins!', 2000),
              doTask('msg3 wins!', 1000)])
.then( function(msg) { console.log(msg); } );
```

```
msg3 wins!
```

Example - Promises and XHR

- Write a program that loads several JSON data sets
- Accumulate all data in one array
- Ensure data is loaded in a specific order so that the data in the resulting dataset also reflects the order
- Use Promises

Promises and XHR



```javascript
// Create a Promise to load a URL
function getURL(url) {
  // Define function to be performed.
  // The Promise will invoke with two functions arguments: resolve and reject.
  // If this work succeeds, invoke resolve( result )
  // If this work fails, invoke reject( error )
  var doWork = function(resolve, reject) {
    // Do the usual XHR stuff
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      // This is called even on 404 etc
      // so check the status
      if (req.status === 200) {
        // Resolve the promise with the response
        var items = JSON.parse(req.response);
        resolve(items);
      }
      else {
        // Otherwise reject with the status text
        // which will (hopefully) be a meaningful error
        reject(Error(req.statusText));
      }
    };

    // Handle network errors
    req.onerror = function() {
      reject(Error("Network Error"));
    };

    // Make the request
    req.send();
  };

  // Return a new promise wrapping doWork.
  return new Promise( doWork );
}
```

http://exploringjs.com/es6/ch_promises.html

10/timer_sequence.js

Example - Promises and XHR

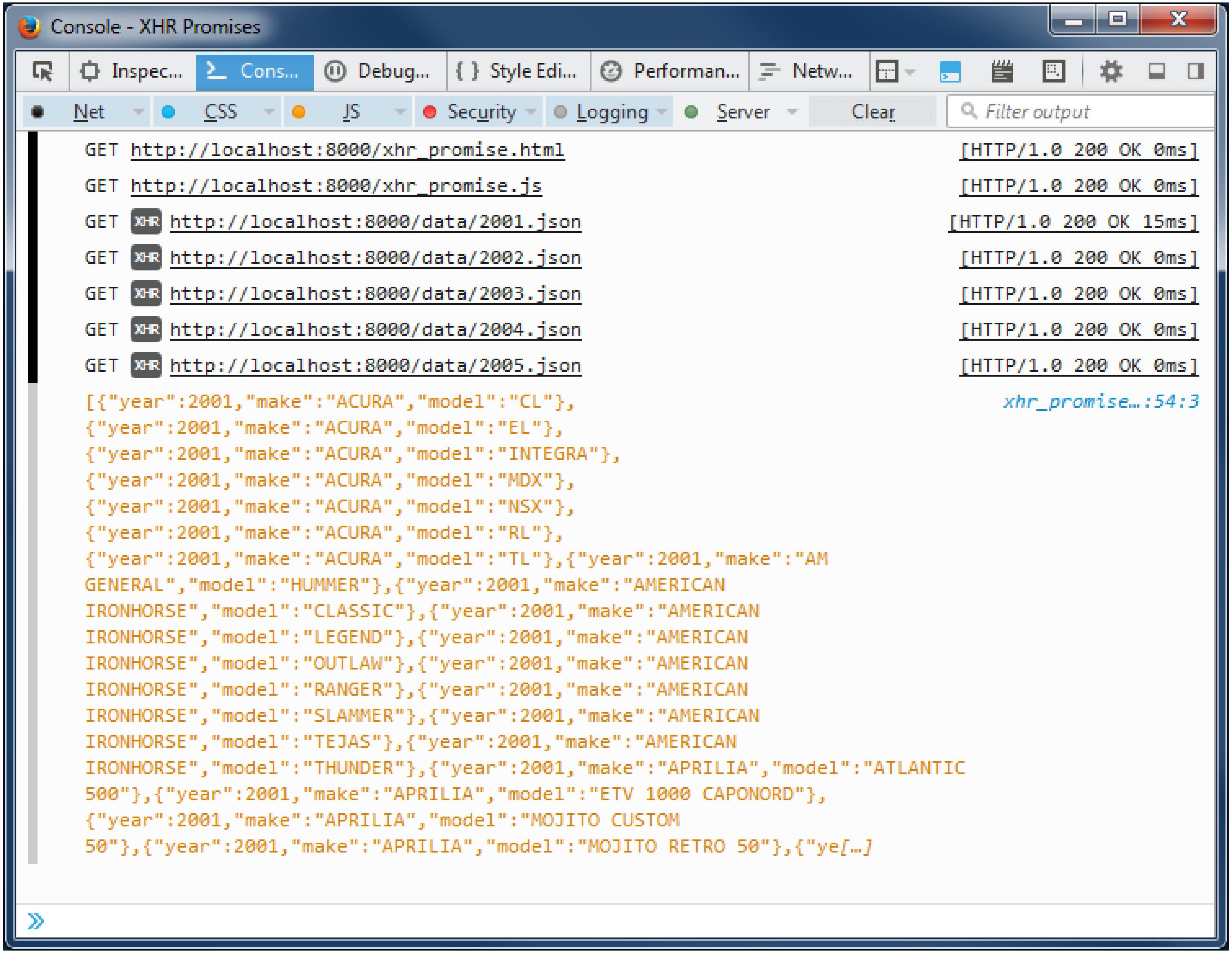
```
// Accumulating data structure
var alldata = [];

// Helper functions that perform accumulation
var accumulate = function(items) {
  alldata = alldata.concat(items);
};

// Display data
var display = function() {
  var s = JSON.stringify(alldata);
  console.log(s);
}

// --------------
// Chaining Promises
function collectall() {
  getURL('data/2001.json')
   .then( accumulate )
   .then( function() { return getURL('data/2002.json'); })
   .then( accumulate )
   .then( function() { return getURL('data/2003.json'); })
   .then( accumulate )
   .then( function() { return getURL('data/2004.json'); })
   .then( accumulate )
   .then( function() { return getURL('data/2005.json'); })
   .then( accumulate )
   .then( display   );
}

// Get started
collectall();
```

Example - Promises and XHR

- To test, start a simple HTTP server
    - Open terminal
    - Change to sample directory
    - Enter…

```
python -m SimpleHttpServer   # For Python 2
python -m http.server        # For Python 3
```

- Open browser
- Visit `http://localhost:8000/xhr_promise.html`



```
GET http://localhost:8000/xhr_promise.html                     [HTTP/1.0 200 OK 0ms]
GET http://localhost:8000/xhr_promise.js                       [HTTP/1.0 200 OK 0ms]
GET XHR http://localhost:8000/data/2001.json                  [HTTP/1.0 200 OK 15ms]
GET XHR http://localhost:8000/data/2002.json                   [HTTP/1.0 200 OK 0ms]
GET XHR http://localhost:8000/data/2003.json                   [HTTP/1.0 200 OK 0ms]
GET XHR http://localhost:8000/data/2004.json                   [HTTP/1.0 200 OK 0ms]
GET XHR http://localhost:8000/data/2005.json                   [HTTP/1.0 200 OK 0ms]
[{"year":2001,"make":"ACURA","model":"CL"},                    xhr_promise…:54:3
{"year":2001,"make":"ACURA","model":"EL"},
{"year":2001,"make":"ACURA","model":"INTEGRA"},
{"year":2001,"make":"ACURA","model":"MDX"},
{"year":2001,"make":"ACURA","model":"NSX"},
{"year":2001,"make":"ACURA","model":"RL"},
{"year":2001,"make":"ACURA","model":"TL"},{"year":2001,"make":"AM
GENERAL","model":"HUMMER"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"CLASSIC"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"LEGEND"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"OUTLAW"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"RANGER"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"SLAMMER"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"TEJAS"},{"year":2001,"make":"AMERICAN
IRONHORSE","model":"THUNDER"},{"year":2001,"make":"APRILIA","model":"ATLANTIC
500"},{"year":2001,"make":"APRILIA","model":"ETV 1000 CAPONORD"},
{"year":2001,"make":"APRILIA","model":"MOJITO CUSTOM
50"},{"year":2001,"make":"APRILIA","model":"MOJITO RETRO 50"},{"ye[…]
```