Server programming with JavaScript

NodeJS

(with all code examples in this archive)

JavaScript everywhere

A common language for frontend and backend:

- Code can be migrated between server and client more easily
- Common data formats (JSON) between server and client
- Common software tools for server and client
- Common testing or quality reporting tools for server and client
- Common dialect between server and client teams

JavaScript in a new context

- JavaScript was traditionally running in a browser context (Browser software is written in low-level languages)
- NodeJS is a new context for executing JavaScript modules (NodeJS is written in C)

Installing NodeJS environment

- At home:

https://github.com/joyent/node/wiki/Installation

- On Amazon EC-2:

http://iconof.com/blog/how-to-install-setup-node-js-on-amazon-aws-ec2-complete-guide/

In the lab (installed):
Open terminal and type:

```
node -v
```

NPM – Node Package Manager for NodeJS

- Installation:

http://howtonode.org/introduction-to-npm

- In the lab (installed):

```
npm help
```

- To see what packages are installed:

```
npm ls
```

Installing libraries with NPM

```
npm install blerg
```

installs the latest version of blerg.

- You can also give install a tarball, a folder, or a url to a tarball.
- If you run *npm install* without additional arguments, it tries to install into the current folder.

Let’s install socket.io

- Create a folder where you are going to put all the examples for node: for example *node*

cd to this folder and

```
npm install socket.io
```

- To check:

```
npm ls
```

Sample programs

- Download sample_code.zip, unzip
- It contains 3 demos:
  - basics
  - simple chat
  - 2-player tic-tac-toe
- There is a separate zipped folder: *node_modules*. Unzip it and copy its content into *simple_chat/js* and *board_games/js* folders, as these two apps use socket.io module.

Hello world, functions, modules

1. BASICS

Hello world

Copy all files from *basics* folder into your *node* folder

We are trying to print:

```
console.log("Hello, world");
```

Basics: Hello world

In file: 01.01.helloworld.js

- rename to helloworld.js
- To run:

```
node helloworld
```

Reminder:
Functions are objects and arguments

```
function say(word)
{
	console.log(word);
}

function execute(someFunction, value)
{
	someFunction (value);
}

execute(say,"Hello");
```

Basics: Functions

In file 01.02.say.js

- rename to say.js
- To run:

```
node say
```

Anonymous functions as arguments

```
function execute(someFunction, value)
{
        someFunction(value);
}


execute
(function(word){console.log(word);},"Hello");
```

Basics: Anonymous functions

In file 01.03.func.js

- rename to func.js
- To run:

```
node func
```

Goodbye, global variables

- There is no more global variables: real encapsulation

- Each variable <u>is visible only inside a module</u>
- To make it publicly visible, we need to export it

Export entire object (single)

```
var obj =
{
        word:"",
        say: function ()
        {
                console.log(this.word);
        }
};

module.exports = obj;
```

Accessing from a test file

```
var myobj = require("./mymodule");
```

Module file name in current directory

```
myobj.word = "Hello module";
myobj.say();
```

Basics: modules

In files:

- 01.04.mymodule.js
- 01.05.moduletest.js

- Run:

```
node moduletest
```

Export multiple objects (functions)

```
var dog ={
        say: function ()    { console.log("Woof"); }
};

function Fish(name, color, length){
        this.name=name;
        this.color=color;

        Fish.prototype.toString = function ()  {
                return "This fish has name:"+this.name +" color:"+this.color;
        }
}

exports.dog = dog;
exports.Fish = Fish;
```

Public name – can be anything

Accessing multiple objects

```
var mymodule = require("./multimodule");

var mycat = mymodule.cat;

mycat.say();

var aquarium = [];
for (var i=0; i<5; i++){
		aquarium.push(new mymodule.Fish("fish "+i, "rgb(0,0,"+(i*10)+")"));
}

for (var i=0; i<aquarium.length; i++)
		console.log (aquarium[i]);
```

Basics: multiple objects in modules

In files:

- 01.06.multimodule.js
- 01.07.multitest.js

- Run:

```
node multitest
```

File I/O

2. BASICS

Reading file: normally

```
var fs = require("fs");


var file1="data/data1.json";
var json1 = fs.readFileSync(file1);


console.log("Read array of size " +
        JSON.parse(json1).length+" from file "+file1);
```

In file: 02.01.readfile.js

Reading 2 files: normally

```
var file1="data/data1.json";
var json1 = fs.readFileSync(file1);
console.log("Read array of size "+JSON.parse(json1).length+"
                                 from file "+file1);
console.log("Reader 1 reports: done");

var file2="data/data2.json";
var json2 = fs.readFileSync(file2);
console.log("Read array of size "+JSON.parse(json2).length+"
                                            from file "+file2);
console.log("Reader 2 reports: done");
```

```
data1.json 636
data2.json 2
```

Reminder:
blocking vs non-blocking I/Os

```
var result = database.query("SELECT * FROM hugetable");
console.log("Hello World");
```

- The program pauses at point when the database layer sends the query to the database.
- That pause can be long: the entire thread is idling, but another request might come in, and if all the threads are busy it will be dropped.
- Threads switching is not free, the more threads we use the more time the CPU spends in storing and restoring the state and the execution stack for each thread takes up memory.
- Simply by using asynchronous, event-driven I/O, Node removes most of this overhead while introducing very little on its own.

Node I/Os

```
database.query ("SELECT * FROM hugetable",
                function(rows) { var result = rows; });
```

callback

```
console.log("Hello World");
```

"When at some point in the future the database server is done and sends the result of the query, then I have to execute the anonymous function that was passed to *database.query()*."

Asynchronous callbacks

- Node asks to think differently about concurrency: callbacks fired asynchronously from an event loop are a simpler concurrency model.
- Node.js continuously cycles through event loop whenever there is nothing else to do, waiting for events. Events like, e.g., a slow database query finally delivering its results.

Reading files Node’s way

data1.json 636
data2.json 2
data3.json 11

```
var file1="data/data1.json";
fs.readFile  (file1,
             function(err,json1) {console.log("Report 1:  from file "+file1);} );

console.log("King fun 1");

var file2="data/data2.json";
fs.readFile  (file2,
             function(err,json2) {console.log("Report 2:  from file "+file2);} );
console.log("King fun 2");

var file3="data/data3.json";
fs.readFile (file3,
             function(err,json3)  {console.log("Report 3: from file "+file3);} );

console.log("King fun 3");
```

02.04.readasynch3.js

Try it out

In files:

- 02.03.readasynch.js
- 02.04.readasynch3.js

Synch vs asynch

```php
<?php
  echo 'Hello';
  sleep(2000);
  echo 'PHP';
  echo 'world';
?>
```

```javascript
console.log('Hello');
setTimeout(
     function(){
       console.log('World');
          }, 2000);
console.log('JavaScript');
```

02.05.test.js

Web server

3. BASICS

Hello world web server

```
var http = require ("http");
var port = 8888;
var httpserver = http.createServer (handleRequest);

function handleRequest(request, response)
{
        response.writeHead (200, {"Content-type":"text/plain"});
        response.write ("Hello from node");
        response.end();
}

httpserver.listen(port);
console.log ("Server listening at port: "+port);
```

Hello world web server (in 9 lines)

Callback function to handle requests

```
var http = require ("http");
var port = 8888;
var httpserver = http.createServer (handleRequest);

function handleRequest (request, response)
{
		response.writeHead (200, {"Content-type":"text/plain"});
		response.write ("Hello from node\n");
		response.end();
}

httpserver.listen(port);
console.log ("Server listening at port: "+port);
```

Let's run it

In file: 03.01.server.js

- To stop server: Ctrl+C

- In browser:

localhost:8888

Or on a command line (in a new terminal window):

```
curl localhost:8888
```

Creating reusable HTTP server module: 03.02.myserver.js

```
var servConnectionManager =
{
…
};
```

Defining content types

```
//file extensions table to serve corresponding content-type to clients
        extensions :    {
                                ".html": "text/html",
                                ".css": "text/css",
                                ".js": "application/javascript",
                                ".png": "image/png",
                                ".gif": "image/gif",
                                ".jpg": "image/jpeg",
                                ".eot":"font/opentype",
                                ".ttf":"font/opentype",
                                ".woff":"font/opentype"
                        },
```

Server *init*

```
init: function ()  //is called when server starts
{
            this.http = require("http");
            this.path = require("path");
            this.filesys = require("fs");

            //create http server
            this.app = this.http.createServer( function(r, s) {
                        servConnectionManager.httpRequestHandler(r,s);
            });

            this.app.listen(this.port);
            console.log ("Server listening at port:" +this.port);
},
```

Handler callback function: on request

```
httpRequestHandler: function (request, response) //serves pages and files on request
{
   // look for a filename in the URL, default to index.html
   var filename = this.path.basename(request.url) || "index.html";
   var ext = this.path.extname(filename);
   var dir = this.path.dirname(request.url).substring(1);
   var localPath =  'public/'; // public folder contains the publicly visible content

   if (servConnectionManager.extensions[ext]) {
      localPath += (dir ? dir + "/" : "") + filename;
      servConnectionManager.path.exists(localPath, function(exists)    {
          if (exists) {
              servConnectionManager.getFile(localPath,
                                          servConnectionManager.extensions[ext], response);
         }
         else  {
             response.writeHead(404);
             response.end();
         }
});}},
```

Serving requested file with header

```
getFile: function (localPath, mimeType, response)
{
    this.filesys.readFile(localPath, function(err, contents)
    {
        if (!err)
        {
            response.writeHead(200,
            {"Content-Type": mimeType,
            "Content-Length": contents.length});
            response.end(contents);
         } else {
            response.writeHead(500);
            response.end();
         }
     }   );
},
```

Finally, export

```
module.exports = servConnectionManager;
```

To try

In file 03.03.serverstart.js:

```
var serverManager=require("./myserver");

serverManager.init();
```

To serve real files

- Create folder public/
- Put some html-css-font-javascript files
- If there is index.html, it can be served from:

localhost:8888

sockets

4. SIMPLE BOARD GAME

Web application: traditional architecture

- Server code written in a scripting language (PHP, Ruby, Python, or JavaScript)
- Server writes and reads data to and from a persistence layer.
- No information is shared between requests except a small amount of session data
- Everything to be remembered is stored in the database (or a key-value store such as MongoDB).

TCP sockets

- All HTTP traffic over the web is transported via TCP sockets.
- The browser *opens a socket* to the server, makes an HTTP request for a resource, waits for it to finish downloading, and then closes the socket.
- After the socket closes, sending any additional data requires opening a new socket.
- If the server has something to tell the client, it needs to wait until the client requests a new resource before it can send data.

Direct interaction between clients: traditional approach

- The server can’t immediately notify one client of something another client does.
- In order to get a timely notification from a server, client has to periodically request information. This is called *polling* because the client periodically polls the server for new information.
- Doing AJAX polling frequently can give the appearance of a pseudo-real-time application in which things happen even when the player isn’t explicitly taking an action.

- Using traditional technologies, we could not maintain an open dialog (data exchange) between the client and the server

Flash sockets

- Flash has a socket support.
- You could use a Flash socket via a loaded flash file that has an interface exposed over a Flash-to-JavaScript.
- The problem with Flash sockets is that they are not secure:
  - They use HTTP port 80 /HTTPS port 443, and thus make normal HTTP requests impossible on these ports: you cannot serve pages when you use these ports for flush sockets.
  - The normal requests need to be made through a nonstandard ports, and this creates problems for clients behind a household or corporate firewall.

Web sockets

- Web Sockets provide a socket-based, real-time, two-way conversation mechanism natively to the browser.

- The idea is to upgrade a standard HTTP socket into a **Web Socket**:
  - Standard handshake technique that both server and client understand.
  - The socket is then **kept open** and allows bidirectional, full-duplex communication between client and server.

WebSocket specification 2009 (www.w3.org/TR/websockets)

Browser support: WebSockets

- Different browsers support different versions of the spec.
- All current-generation browsers except IE9 have some version of WebSockets turned on.
- Because of proxies, caches, and IE9, you can’t use standard WebSockets without some *fallback*.

Fallback with socket.io

- Instead of using straight WebSockets, there is a Node.js library called **socket.io** that provides a consistent client and server API regardless of whether native WebSockets or one of the fallback mechanisms are supported.
- Socket.io abstracts WebSockets and supported fallbacks on both the client and the server side: Socket.IO will use feature detection to decide if the connection will be established with WebSocket, AJAX long polling, Flash, etc

Supported transports

Socket.IO selects the most capable transport at runtime, without it affecting the API.

- WebSocket
- Adobe® Flash® Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

Decreasing speed, so it selects the top available

Features of socket.io

- Capability to transparently send JSON data over sockets
- Support for any number of custom events
- Integration with NodeJS servers: you can use a single app to serve your HTTP methods, your WebSockets, and your static files.
- Support for heartbeats, timeouts, and disconnection

Server side

- Socket.io works by listening for **connection** events.
- These events trigger a callback with a socket object as a parameter.
- You can then attach additional listeners for both standard events (such as disconnect) and custom named events.
- To send data you call **socket.emit** with a name for the event and any data that needs to be passed along.
- You can send events to all connected sockets except the socket itself by calling **socket.broadcast.emit**.

Server code: socket listening over http server

```
var httpserver= require('http').createServer(requestHandler )  ;

app.listen(8888);

function requestHandler (req, res) {
…
 }

//create socket listener
socket = require("socket.io");
socketio = socket.listen(httpserver);
socketio.sockets.on       ('connection', function (socket)
                                   {socketEventHandler(socket);}
                          );
```

Server code: socketRequestHandler

```
function socketEventHandler (socket)
{
      socket.on('req_add_player', function ( name)
      {
         var result =app.addPlayer(name);
         socketio.sockets.socket(socket.id).emit('resp_added_player',result );

      });

      socket.on('req_start_game', function ( )
      {
         var result  = app.initGame();
         socketio.sockets.emit('resp_update', result);
      });
}
```

Emits JSON object to a particular client

Broadcasts JSON object to all connected clients

Client side

- You need to include a special JavaScript file in the head of HTML:

```
<script src="/socket.io/socket.io.js"></script>
```

- This is a path created by socket.io on server.
- It also automatically determines which transport mechanism to use: straight WebSockets or one of the fallbacks.
- Because the socket.io.js file was pulled from the same server as the socket is being connected to, you don’t need to provide a URI or port to connect to. (You can provide these if necessary to connect to a different server.)

Client code: reference

In HTML head:

```
<script src="/socket.io/socket.io.js"></script>
```

Client: JavaScript

```
socket = io.connect();
//listens
socket.on('resp_updated_game',
          function(currentPlayerID, state, gameTable)
         {   gameState.state = state;
             gameState.currentPlayerID = currentPlayerID;
             gameState.gameTable = gameTable;
             gameState.updateHTML();
         });


//onclick
gameState.changeCellValue (x,y);
socket.emit('req_game_update', gameState.currentPlayerID, gameState.state,
gameState.gameTable);
```

Using socket.io for a simple chat

- Working code is in *chat* folder
- Copy node_modules folder (containing socket.io module) into this folder
- From this folder: to start server

```
node chat.js
```

- To test chat - in two different browsers ask for:

localhost:8888

Tic-tac-toe: simple 2-player game

- Working code is in ttt folder
- Copy node_modules folder (containing socket.io module) into this folder
- From this folder: to start server

```
node start
```

- To test game: open localhost:8888 in two different browsers

Client code: listening for server data

```
init: function ()
{
        //establish connection
        this.socket = io.connect();

        //set up socket listeners
        this.socket.on (“custom message title“,
                        function (args) { what to do with args}
        );
}
```

Client code: handling user events

- Click on login button (onclick=“requestlogin”):

//ends up in roomBrowser panel on success, or the same login screen with errors

```
function requestLogin ()        {
        var userName = document.getElementById("username").value;
        if (!userName)
                alert ("Enter your name to login");
        var userData = {"username":userName};

        this.socket.emit('req_login',userData);
}
```

Emits request for login

Server side

```
//Needs to listen for "req_login" event
socket.on('req_login', function (clientData){
        var newuser = appRoomBrowser.getUser(clientData);

        //resp_login - to a specific client
        socketRouter.sockets.socket(sessionID).emit('resp_login',
                newuser, appRoomBrowser.getRooms());
                }
        });
```

Emits response – to the same user

Server: setting up socket

Need an HTTP server

```
socketlibrary = require("socket.io");
var socketRouter = socketlibrary.listen(httpserver);
socketRouter.set('log level', 1)
socketRouter.sockets.on ('connection', socketEventHandler );
```

Name of a callback function

Distinguishing clients: unique socket identifier

```
function socketEventHandler (socket)
{
        var sessionID = socket.id;
        console.log("CLIENT CONNECTED with
                                        socket.id="+sessionID);

        sessions[sessionID]=-1; //guest id assigned

        //After login:
        sessions[sessionID] = newuser;
```

Parameter of every socket event

Talking to a specific client

```
socketRouter.sockets.socket(sessionID).emit
                    ('resp_session_id_assigned', sessionID);

socketRouter.sockets.socket(sessionID).emit
          ('resp_login', newuser, appRoomBrowser.getRooms());
```

Sending message to all clients

```
socketRouter.sockets.emit
        ('resp_browser_update',
        appRoomBrowser.getRooms());
```

Virtual socket rooms

- Joining the room

```
socket.join(room.alias);
```

- Notifying everyone in the room

```
socketRouter.sockets.in(room.alias).emit
                         ('resp_room_update',room);
```

Leaving rooms and closing the room

- when all disconnected, room will be cleared automatically

```
for(var i = 0; i < room.players.length; i++) {
        socketRouter.sockets.socket
                (room.players[i].sessionID).disconnect();
}
```

“Disconnect” event

```
socket.on('disconnect', function () {
        var user=sessions[sessionID];
        //find if he was a part of a room
        var roomAlias=user.assignedRoomAlias;
        if (roomAlias) {
                disconnect him from this room
                //notify  everyone in the room
                socketRouter.sockets.in(room.alias).emit
                    ('resp_room_deleted',appRoomBrowser.getRooms());
                delete rooms[roomAlias];
        }
        //clear sessions
        delete sessions[sessionID];
        console.log("DISCONNECTED USER username="+user.username
                                                  +" session="+sessionID);
```

Room browser and handling user logins

- Working code in board_games.zip

- To start a server– you need to move into board_games folder, open terminal and:

```
node js/start
```

- Then you can connect with multiple clients from

localhost:8888