# Node.js: JavaScript on the server

Claudia Hauff
cse1500-ewi@tudelft.nl

TU Delft

# Learning goals

- **Explain** the main ideas behind `Node.js`
- **Implement** basic network functionality with `Node.js`
- **Explain** the difference between `Node.js`, `npm` & `Express`
- **Create** a fully working Web application that has client- and server-side interactivity
- **Implement** client/server bidirectional communication through WebSockets

# What is Node.js?

# Node.js in its own words …

"… intended for C++ programmers who want to embed the V8 JS engine within a C++ application."

"Node.js® is a platform **built on Google Chrome's JavaScript runtime** for easily building **fast**, **scalable network** applications.

Node.js uses an **event-driven**, **non-blocking** **I/O** model that makes it lightweight and efficient, perfect for **data-intensive real-time applications** that run across distributed devices."

# History of Node.js

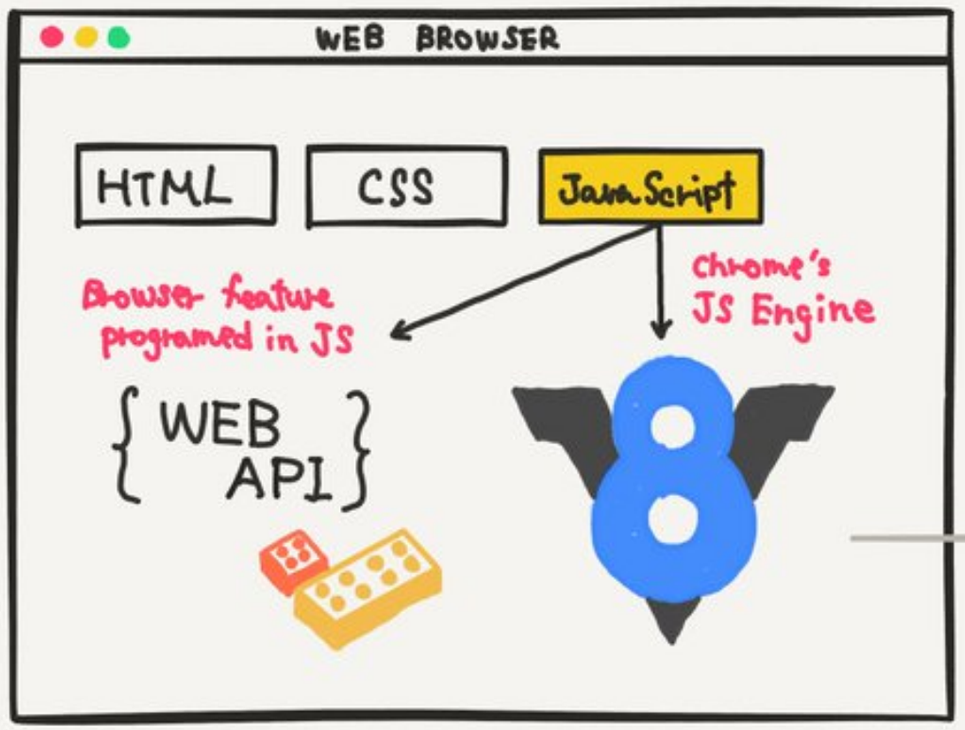
- 2008: Google's **V8** engine is open-sourced.
- 2009: Node.js is released. It builds on V8.
- 2011: Node.js' package manager (npm) is released.
- December 2014: io.js is forked.
- May 2015: io.js merges back with Node.js. The **Node.js Foundation** steers the development.
- 2017: Node.js becomes a **first-class citizen of V8**.
- August 2018: Node.js has been downloaded 1 billion times.

drawing by Mariko Kosaka (@kosamari)

@kosamari
March 3, 2017

# What is Node.JS ??

What's browser JS ??

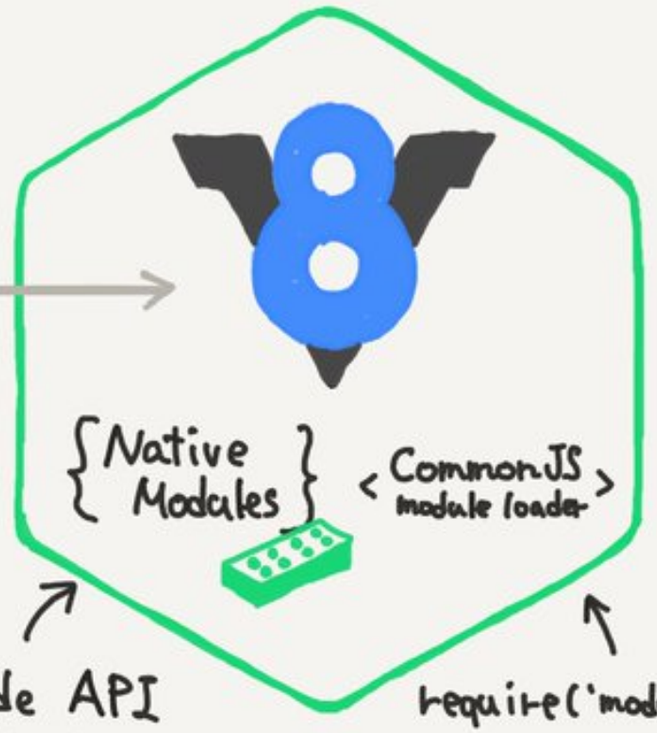



Web Platform
- Window
- Fetch
- Web Audio etc...

Language features
- ES6
- Promise
- Typed Array etc...

Web API & JavaScript Language are often described as the JavaScript (in intro books)

Node is a JavaScript ENVIRONMENT with Special API (like http) and default module loader.

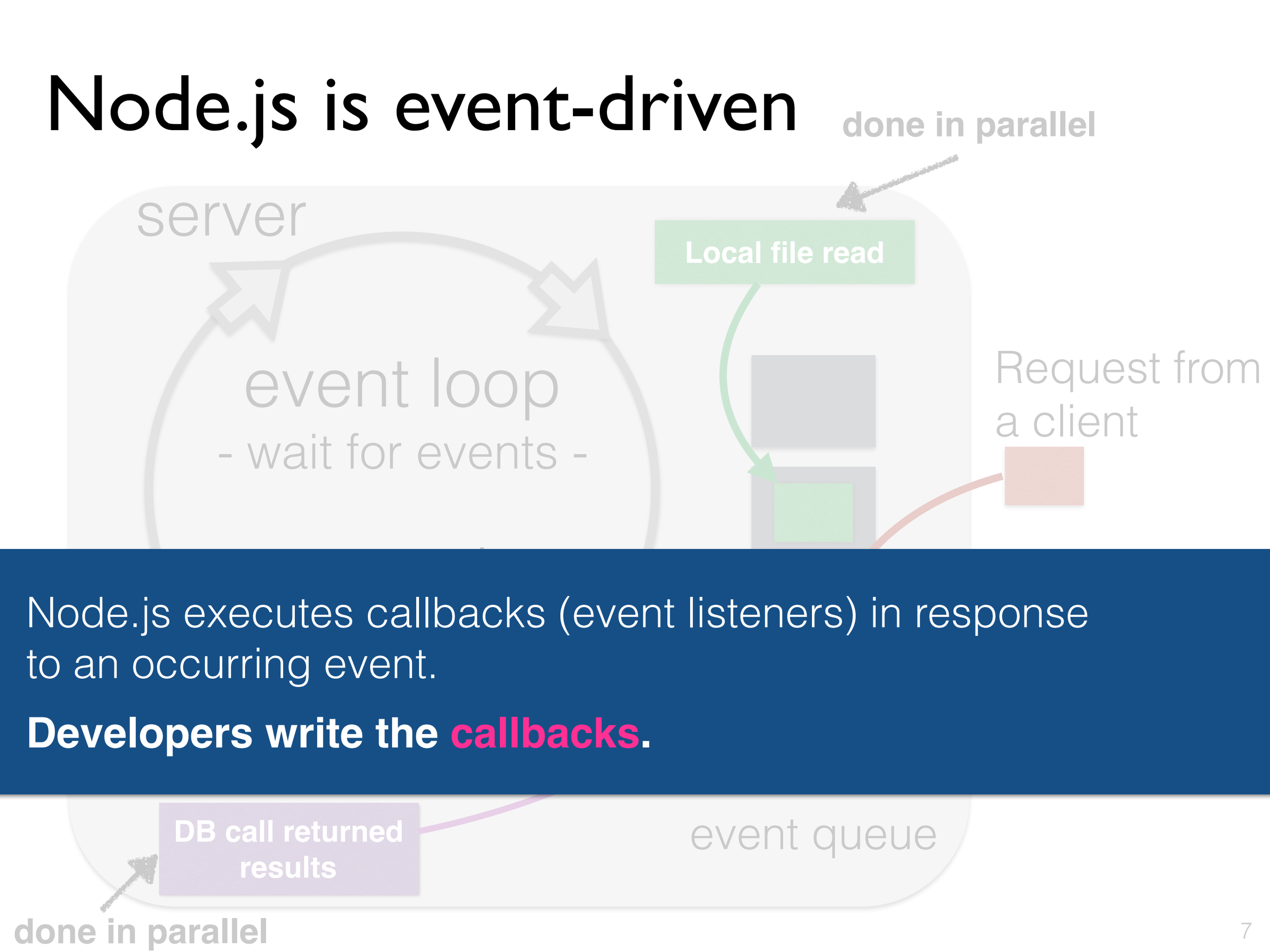



Node API
- http
- fs
- process etc...

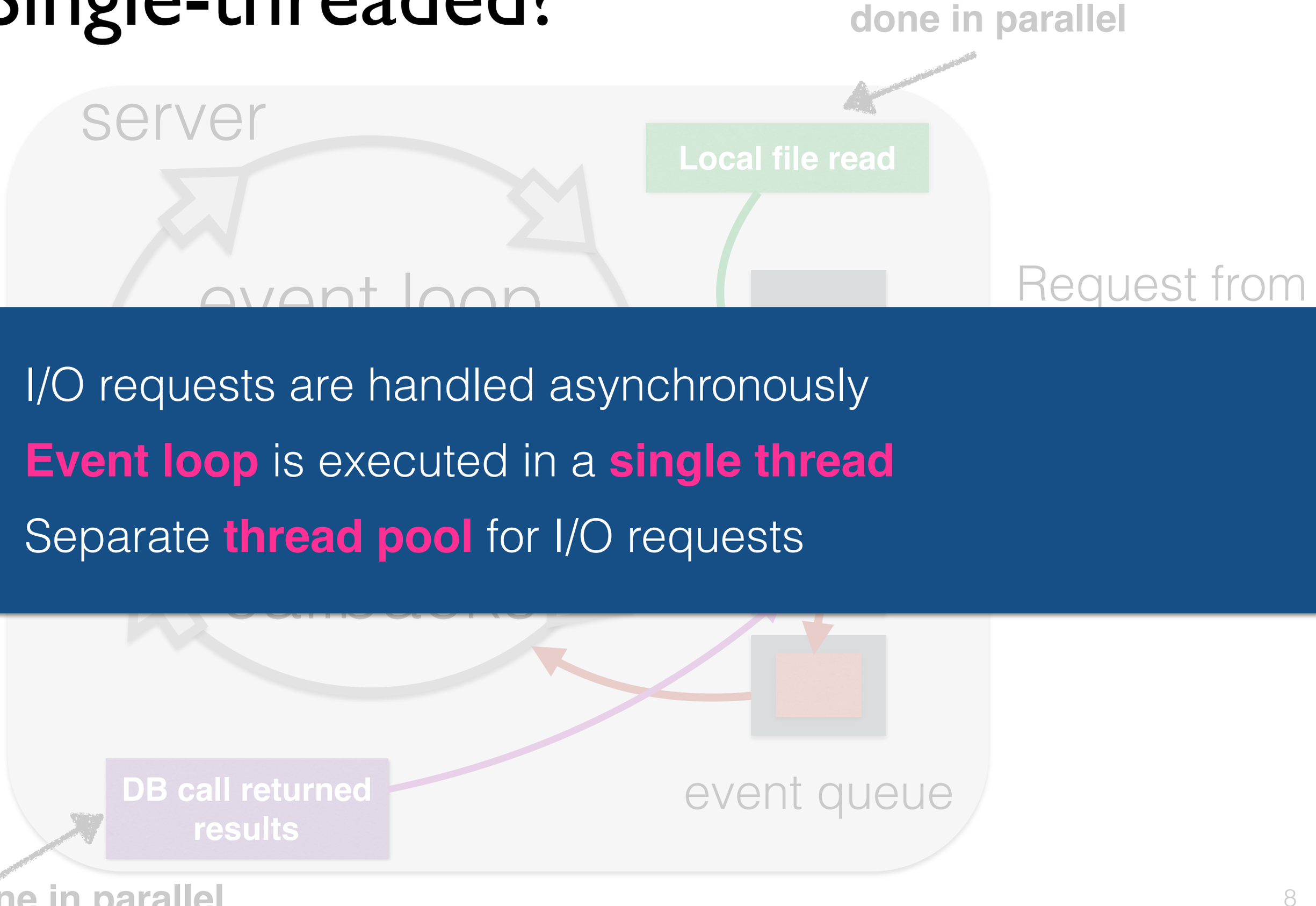



Node runs on EVERYWHERE.

# Node.js is event-driven



# Single-threaded?




- I/O requests are handled asynchronously
- **Event loop** is executed in a **single thread**
- Separate **thread pool** for I/O requests

# Async is the default

done in parallel

server

Local file read

```
fs.readFile(‘file','utf8', function(err, data) {
    if(err) throw err;
    console.log(data);
});
```

callback: executed when file reading is complete

```
let data = fs.readFileSync(‘file','utf8');
```

execute

DB call returned results

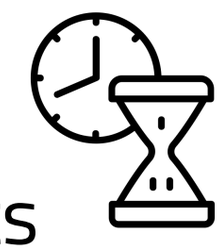
event queue

done in parallel

# Node.js: single-threaded but highly parallel

- **I/O bound programs**: programs constrained by data access (adding more CPUs or main memory will not lead to large speedups)

- Many tasks might require **waiting time**
  - Waiting for a database to return results
  - Waiting for a third party web service
  - Waiting for connection requests

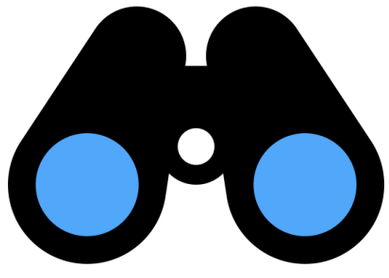


**`Node.js` is designed with these use cases in mind.**

# Node.js in examples

Demo time: example 1

# Watching a file for changes …



# Watching a file for changes

`require()` **usually** returns a JavaScript **object**

Node.js `fs` **module***

**Self-contained** piece of code that provides reusable functionality.

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

```
var file = process.argv[2];
//read the file name from the command line

fs.watch(file, function() {
  console.log("File changed!");
});

console.log("Now watching " + file);
```

Executed immediately after the **setup** of the callback

**Assumption**:
file to watch exists!

* **module**: any file/directory that can be loaded by `require()`
  **package**: any file/directory described by a `package.json` file
  (most npm packages are modules)

# Watching a file for changes

`require()` **usually** returns a JavaScript **object**

Node.js `fs` **module***

**Self-contained** piece of code that provides reusable functionality.

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

var file = process.argv[2];
// Get the file name from the command line

fs.watch(file, function() {
  console.log("File changed!");
});

console.log("Now watching " + file);
```

**polls** file for changes

**callback**: defines what should happen when the file changes

**anonymous function**

**asynchronous**

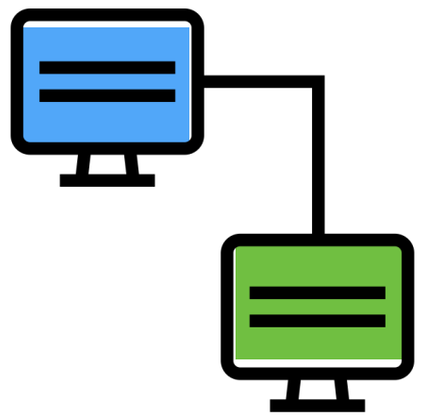
Executed immediately after the **setup** of the callback

**Assumption**:
file to watch exists!

* **module**: any file/directory that can be loaded by `require()`
  **package**: any file/directory described by a `package.json` file (most npm packages are modules)

Demo time: example 2

# Low-level networking with Node.js



# Not just for web programming …

- Built for **networked programming**
- Node.js has built-in support for **low-level** socket connections
- TCP socket connections have **two endpoints**
  1. **binds** to a numbered port
  2. **connects** to a port

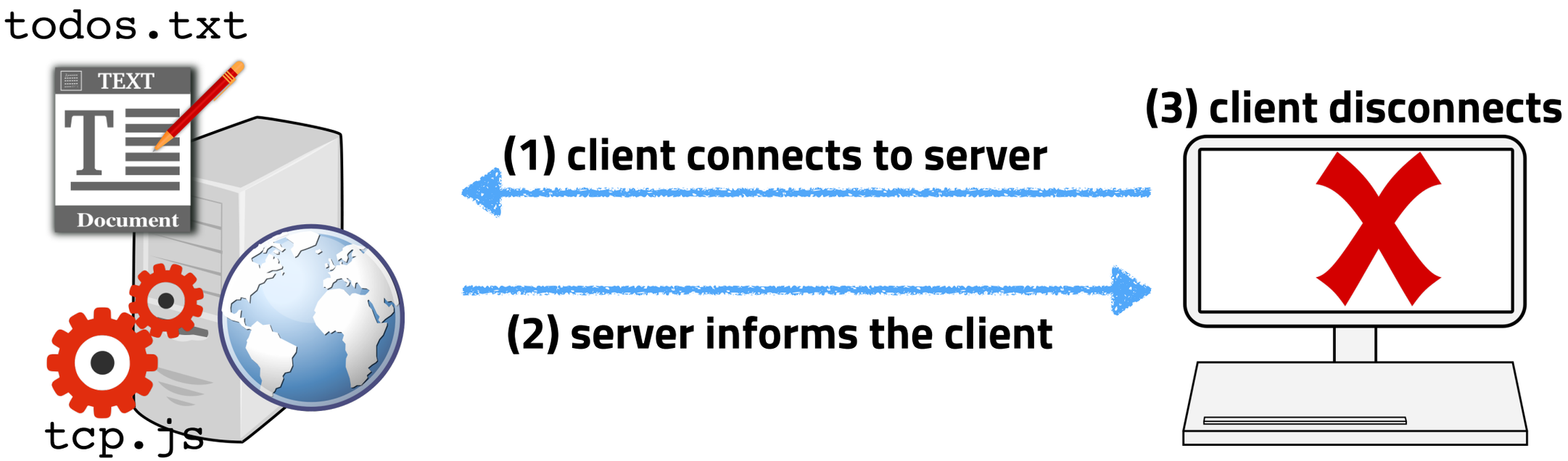
**Analogous example: phone lines.**

One phone binds to a phone number.

Another phone tries to call that phone.

# Client / server communication

**Task**: Server informs connected clients about changes to file `todos.txt`.




```
server:~ $
Listening for subscribers …
Subscriber connected.
Subscriber disconnected.
```

```
client:~ $
Now watching todos.txt for
changes …
File todos.txt changed: …
```

# Callbacks all the way …

**Task**: Server informs connected clients about changes to file `todos.txt`.

**Client:** `telnet localhost <port>`

```
const fs = require("fs");
const net = require("net");

const filename = process.argv[2];
const port = process.argv[3];

var server = net.createServer(function(connection) {
  //connection object used for data transfer
});

server.listen(port, function() {
  console.log("Listening to subscribers...");
});
```

bind to a port

# Callbacks all the way …

**Task**: Server informs connected clients about changes to file `todos.txt`.

**Client:** `telnet localhost <port>`

```
const fs = require("fs");
const net = require("net");

const port = process.argv[2];
const file = process.argv[3];

var server = net.createServer(function(connection) {
  //connection object used for data transfer
});

server.listen(port, function() {
  console.log("Listening to subscribers...");
});
```

server **object** is returned

**callback** function is invoked when another endpoint connects

**bind to a port**

Demo time: example 3

# Our first *Hello World!* web server

Node.js is not a Web server. It provides functionality to implement one!

# The “Hello World” of Node.js

node.js http module

HTTP **request** object

HTTP **response** object

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

var port = process.argv[2];

var server = http.createServer(function(req, res) {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Hello World!");
  console.log("HTTP response sent");
});

server.listen(port, function() {
  console.log("Listening on port " + port);
});
```

**Create** a **HTTP response** & send it

Start the **server** on the command line: **`$ node web.js 3000`**
Open the browser (**client**) at: http://localhost:3000

# The “Hello World” of Node.js

node.js http module

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

var port = process.argv[2];

var server = http.createServer(function(req, res) {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Hello World!");
  console.log("HTTP response sent");
});

server.listen(port, function() {
  console.log("Listening on port " + port);
});
```

create a web server

**A callback:** what to do if a request comes in

Start the **server** on the command line: **`$ node web.js 3000`**
Open the browser (**client**) at: http://localhost:3000

# A little refactoring …

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

var port = process.argv[2];

function simpleHTTPResponder(req, res) {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Hello World!");
  console.log("HTTP response sent");
}

var server = http.createServer(simpleHTTPResponder);

server.listen(port, function() {
  console.log("Listening on port " + port);
});
```

function as parameter

Untitled-1 — Web-Teaching

EXPLORER

OPEN EDITORS

Untitled-1

WEB-TEACHING

- .vscode
- demo-code
  - balloons-game
    - node_modules
    - public
    - routes
    - tests
    - views
    - .eslintrc.js
    - app.js
    - game.js
    - package-lock.json
    - package.json
    - statTracker.js
  - balloons-wireframes
  - minimum-example-ws
  - node-ajax-ex
  - node-component-ex
  - node-cookies-ex
  - node-ejs-ex
  - node-express-ex
  - node-file-watching-ex
  - node-sessions-ex
  - node-tcp-ex
  - node-url-routing-ex
  - node-web-ex
  - node-websocket-ex
  - README.md
- img
- slides
- .gitignore
- .travis.yml
- Assignment-1.md
- Assignment-2.md
- Assignment-3.md

OUTLINE

Untitled-1 ×

1

Demo time: example 4

master 0 0 Ln 1, Col 1 Spaces: 4 UTF-8 LF

# Using URLs for routing

```
var http = require("http");
var url = require("url");

var port = process.argv[2];

function simpleHTTPResponder(req, res) {
  //parse the URL
  var uParts = url.parse(req.url, true);

  //implemented path
  if (uParts.pathname == "/greetme") {
    res.writeHead(200, { "Content-Type": "text/plain" });

    //parse the query
    var query = uParts.query;
    var name = "Anonymous";

    if (query["name"] != undefined) {
      name = query["name"];
    }

    res.end(" Greetings " + name);
  }
  //all other paths
  else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("Only /greetme is implemented.");
  }
}

var server = http.createServer(simpleHTTPResponder);

server.listen(port, function() {
  console.log("Listening on port " + port);
});
```

otherwise send back a **404 error**

# Using URLs for routing

```
var http = require("http");
var url = require("url");

var port = process.argv[2];

function simpleHTTPResponder(req, res) {
  //parse the URL
  var uParts = url.parse(req.url, true);

  //implemented path
  if (uParts.pathname == "/greetme") {
    res.writeHead(200, { "Content-Type": "text/plain" });

    //parse the query
    var query = uParts.query;
    var name = "Anonymous";

    if (query["name"] != undefined) {
      name = query["name"];
    }

    res.end(" Greetings " + name);
  }
  //all other paths
  else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("Only /greetme is implemented.");
  }
}

var server = http.createServer(simpleHTTPResponder);

server.listen(port, function() {
  console.log("Listening on port " + port);
});
```

if the **pathname** is **`/greetme`** we greet

we can **extract params** from the **URL**

otherwise send back a **404 error**

Low-level Node.js capabilities are important to know about (you don't always need a Web server), but ...

... it is tedious to write an HTTP server this way.

How do you send CSS files and images?

# Express

# Express

- Node.js has a **small core** code base
- Node.js comes with some **core modules included**
- Express is not one of them (but we have **NPM**)

```
$ cd my-project-folder
$ npm init -y
$ npm install express --save
```

Demo time: example 5

"The Express module creates a **layer on top of the core http module** that handles a lot of **complex things** that we don't want to handle ourselves, like **serving up static** HTML, CSS, and client-side JavaScript files." **(web course book, Ch. 6)**

# Packages do not have to do a lot

npm

find packages

**Easy sharing.** Manage teams and permissions with one click. Create a free org »

★ smallest public

> Find the smallest number in a list

## Install

```
$ npm install --save smallest
```

### Stats

**41** downloads in the last day

**354** downloads in the last week

**1,241** downloads in the last month

No open issues on GitHub

No open pull requests on GitHub

5 lines (4 sloc) | 135 Bytes

```
1  'use strict'
2  module.exports = function smallest (values) {
3    return Math.min.apply(Math, Array.isArray(values) ? values : arguments)
4  }
```

Untitled-1 — Web-Teaching

EXPLORER

OPEN EDITORS

Untitled-1

WEB-TEACHING

- .vscode
- demo-code
  - balloons-game
    - node_modules
    - public
    - routes
    - tests
    - views
    - .eslintrc.js
    - app.js
    - game.js
    - package-lock.json
    - package.json
    - statTracker.js
  - balloons-wireframes
  - minimum-example-ws
  - node-ajax-ex
  - node-component-ex
  - node-cookies-ex
  - node-ejs-ex
  - node-express-ex
  - node-file-watching-ex
  - node-sessions-ex
  - node-tcp-ex
  - node-url-routing-ex
  - node-web-ex
  - node-websocket-ex
  - README.md
- img
- slides
- .gitignore
- .travis.yml
- Assignment-1.md
- Assignment-2.md
- Assignment-3.md

OUTLINE

Untitled-1 ×

1

Demo time: Example 6!

Hello World in Express!

master 0 0

Ln 1, Col 1 Spaces: 4 UTF-8 LF

# "Hello World" of Express

```
var express = require("express");
var url = require("url");
var http = require("http");

var port = process.argv[2];
var app = express();
http.createServer(app).listen(port);

var htmlPrefix = "<!DOCTYPE html><html><head></head><body><h1>";
var htmlSuffix = "</h1></body></html>";

app.get("/greetme", function(req, res) {
  var query = url.parse(req.url, true).query;
  var name = query["name"] != undefined ? query["name"] : "Anonymous";

  res.send(htmlPrefix + "Greetings " + name + htmlSuffix);
});

app.get("/goodbye", function(req, res) {
  res.send(htmlPrefix + "Goodbye to you too!" + htmlSuffix);
});

app.get("/*", function(req, res) {
  res.send("Not a valid route ...");
});
```

**error-prone**, **not maintainable**, fails at anything larger than a toy project.

**Express creates HTTP headers for us**

# "Hello World" of Express

```
var express = require("express");
var url = require("url");
var http = require("http");

var app = express();
http.createServer(app).listen(port);

var htmlPrefix = "<!DOCTYPE html><html><head></head><body><h1>";
var html                                    dy></html>";

app.get("/greetme", function(req, res) {
  var query = url.parse(req.url, true).query;
  var name = query["name"] != undefined ? query["name"] : "Anonymous";

  res.send(htmlPrefix + "Greetings " + name + htmlSuffix);
});

app.get("/goodbye", function(req, res) {
  res.send(htmlPrefix + "Goodbye to you too!" + htmlSuffix);
});

app.get("/*", function(req, res) {
  res.send("Not a valid route ...");
});
```

app object is our way to use Express' abilities

URL "route" set up

**error-prone**, **not maintainable**, fails at anything larger than a toy project.

another route

all other routes

Express creates HTTP headers for us

# Express and its static file server

- **Static files**: files that are not created/changed on the fly
  - CSS, JavaScript (client-side), HTML, Images, video, etc.
- A single line of code is sufficient to serve static files:

```
app.use(express.static(__dirname + "/static"));
```

globally available string containing the name of the current module

a static files are contained in this directory

- Express always **first** checks the static files for a given route - if not found, the dynamic routes are checked

# How to build a web application

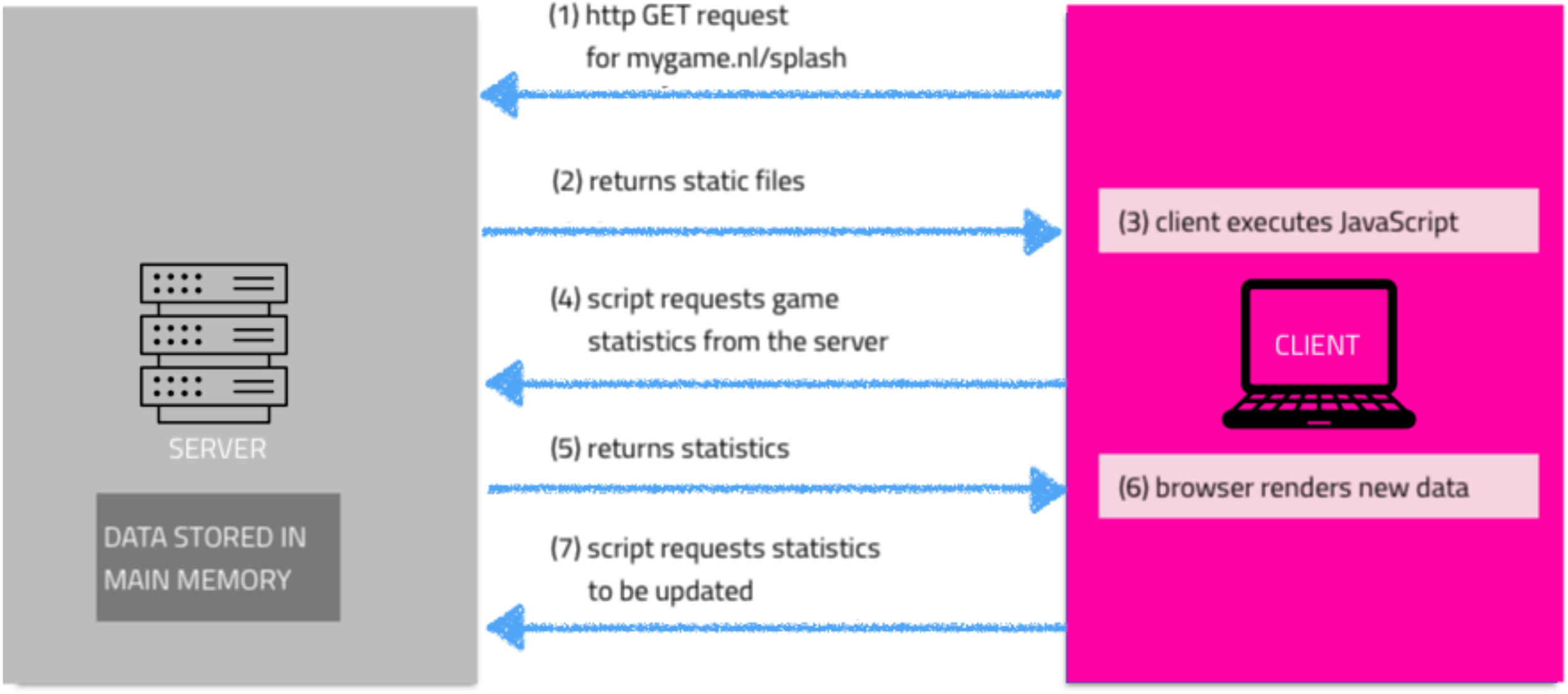
# Development strategy

1. Develop the **client-side code** (HTML, CSS, JavaScript)
2. Place all files into some directory (e.g. `/client`) **on the server**
3. Write the **Node.js server code** using Express
4. Set **Express' static file path** to the directory of step 2
5. Add interactivity between client and server

```
app.js
client/
    index.html
    html/
        =>game.html
        =>splash.html
    images/
        =>background.png
        =>logo.png
css/
    =>layout.css
    =>style.css
javascript/
    =>client-app.js
```

Boilerplate structure and code: code generators help (A5)!

# Typical web app flow




Course book: client-server interaction via Ajax. Useful for our splash screen. We'll use WebSockets for the actual game play.

# JSON: exchanging data between the client and server

# XML vs. JSON

```xml
<!--?xml version="1.0"?-->
<timezone>
  <location></location>
  <offset>1</offset>
  <suffix>A</suffix>
  <localtime>20 Jan 2014 02:39:51</localtime>
  <isotime>2014-01-20 02:39:51 +0100</isotime>
  <utctime>2014-01-20 01:39:51</utctime>
  <dst>False</dst>
</timezone>
```

```json
{
  "timezone": {
    "offset": "1",
    "suffix": "A",
    "localtime": "20 Jan 2014 02:39:51",
    "isotime": "2014-01-20 02:39:51 +0100",
    "utctime": "2014-01-20 01:39:51",
    "dst": "False"
  }
}
```

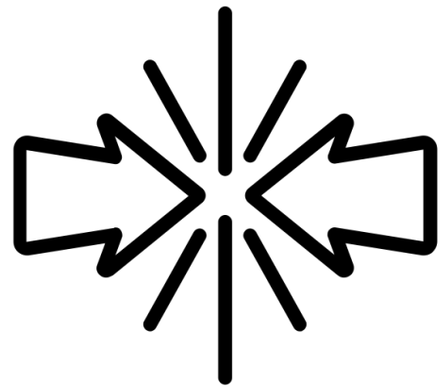
# Exchanging data

- Ten/twenty years ago **XML** was the standard data exchange format - well defined, but not easy to handle

- XML is often too **bulky**; JSON has a smaller footprint

- JSON to JS object and the reverse is possible using **built-in JavaScript** functions
  - `JSON.parse, JSON.stringify`

# JSON vs. JavaScript objects



- JSON: all property names must be enclosed in **quotes**

- Objects created from JSON **do not have functions** as properties (functions are stripped in the conversion to JSON)

- Express has a dedicated response object method to send JSON: `res.json(param)`

# Ajax: dynamic updating on the client

# What is Ajax?

- Asynchronous JavaScript and XML: XML is in the name only
- Ajax enables dynamic loading of content **without refetching the entire page**
- Ajax is a technology that **injects** new data into an existing document
- `jQuery` **hides a lot of complexity** and makes Ajax calls easy

Nederlands Sign in

Bing

chess co

chess com

chess computer

chess connects us

chess corner

chess com news

chesscool

chess computers test

chess combinations preparing

Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility What's New

Filter URLs | All HTML CSS JS XHR Fonts Images Media WS Other | Persist Logs | Disable Cache | No Thro

| Status | Met... | Domain | File | Cause | Type | Transferred | Size |
|---|---|---|---|---|---|---|---|
| 200 | GET | www.bing.... | Suggestions?pt=page.home&mkt=nl-nl... | xhr | html | 1.07 KB | 2.62 ... |
| 200 | POST | www.bing... | ls.gif?IG=539D44B6FFB541CDACD68A... | beacon | gif | 229 B | 0 B |
| 200 | GET | ww... | | | | ...8 KB | 2.66 ... |
| 200 | GET | ww... | | | | ...7 KB | 2.60 ... |
| 200 | GET | www.bing.... | Suggestions?pt=page.home&mkt=nl-nl... | | html | 1.08 KB | 2.71 KB |
| 200 | GET | www.bing.... | Suggestions?pt=page.home&mkt=nl-nl... | xhr | html | 1.12 KB | 2.81 KB |
| 200 | GET | www.bing.... | Suggestions?pt=page.home&mkt=nl-nl... | xhr | html | 1.10 KB | 2.80 ... |
| 204 | POST | www.bing.... | lsp.aspx | xhr | xml | 2.58 KB | 0 B |
| 204 | POST | www.bing.... | lsp.aspx | xhr | xml | 2.24 KB | 0 B |

XMLHttpRequest

61 requests | 890.97 KB / 451.02 KB transferred | Finish: 18.45 s | DOMContentLoaded: 133 ms | load: 223 ms

Headers Cookies Params Response Timings Stack Trace

Preview

- chess com
- chess computer
- chess connects us
- chess corner
- chess com news

Response Payload

```
1  <ul class="sa_drw" id="sa_ul" data-priority="2"><li class="sa_
2  sa_inst.setData({"Inst":{"IG":"88BB0264595741C79B9D36C6CCB40E1
3  //]]></script>
```

# On the client: HTML & JS

jQuery way

```
<!DOCTYPE html>
<head>
    <title>Plain text TODOs</title>
    <script src="http://code.jquery.com/jquery-3.3.1.min.js" type="text/javascript"></script>
    <script src="js/client-app.js" type="text/javascript"></script>
</head>

<body>
    <main>
        <section id="todo-section">
            <p>My list of TODOS:</p>
            <ul id="todo-list">
            </ul>
        </section>
    </main>
</body>
</html>
```

Load the JavaScript files, **start with jQuery.**

Define where the TODOs will be added. Empty list.

**Callback**: define what happens when a todo object is available.

Dynamic insert of list elements into the DOM.

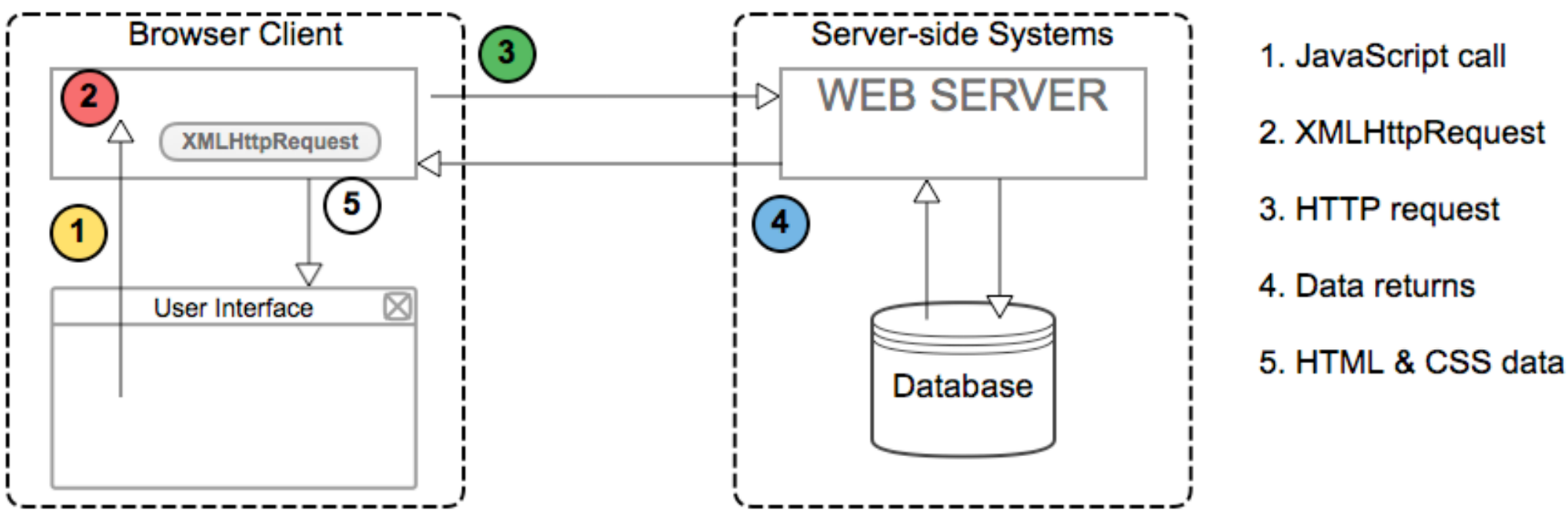
when the document is loaded, execute `main()`.

```
var main = function() {
  "use strict";

  var addTodosToList = function(todos) {
    console.log("Loading todos from server");
    var todolist = document.getElementById("todo-list");
    for (var key in todos) {
      var li = document.createElement("li");
      li.innerHTML = "TODO: " + todos[key].message;
      todolist.appendChild(li);
    }
  };

  $.getJSON("../todos", addTodosToList)
    .done(function() {
      console.log("Ajax request successful.");
    })
    .fail(function() {
      console.log("Ajax request failed.");
    });
};
$(document).ready(main);
```

Ajax

# Ajax: how does it work?

1. Web browser creates a **`XMLHttpRequest`** object
2. `XMLHttpRequest` **requests data** from a Web server
3. **Data is sent back** from the server
4. On the client, **JavaScript code injects the data** into the page



# WebSockets

# WebSockets

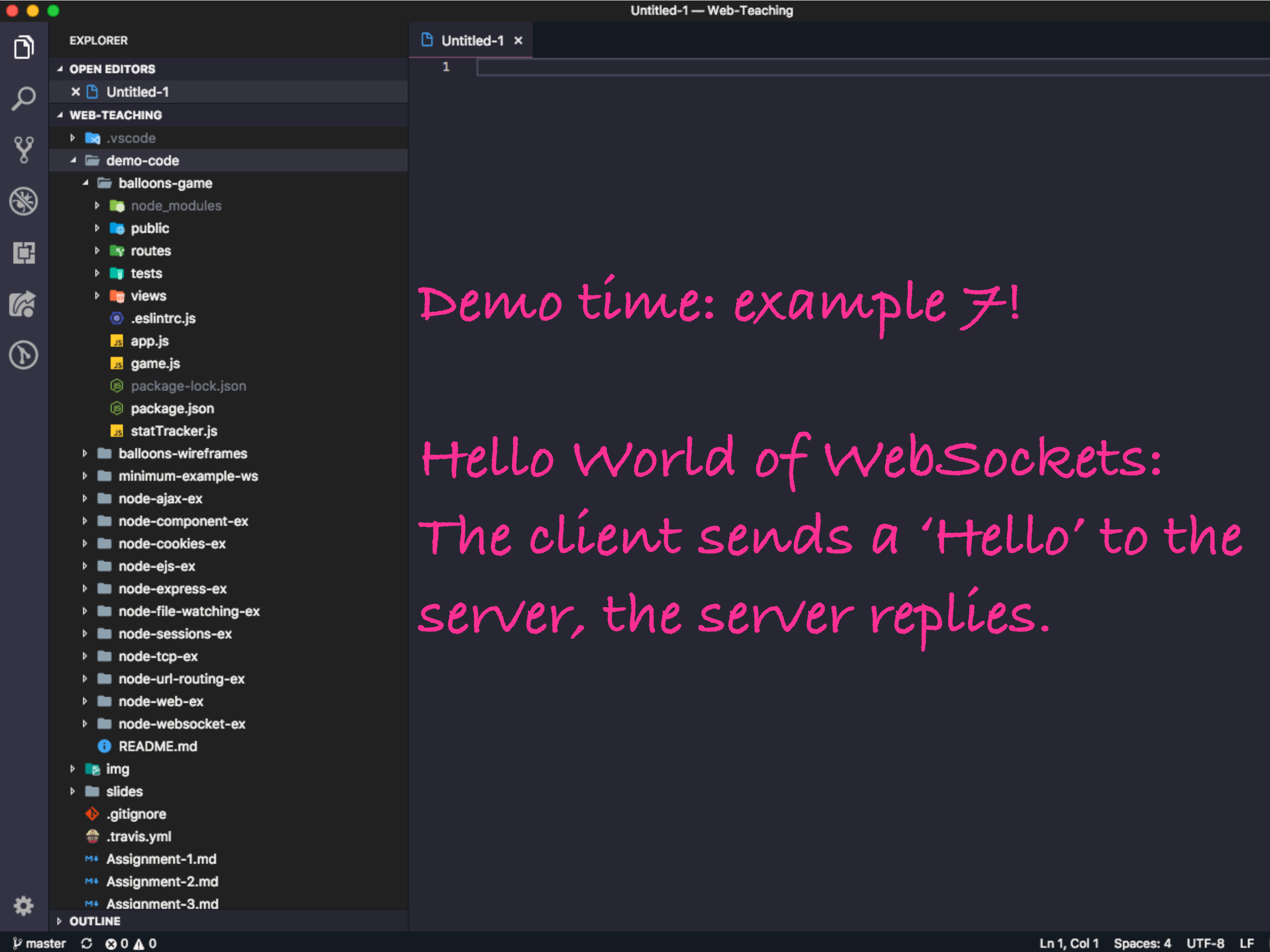
- **Bidirectional communication** between client and server

- **Protocol upgrade** initiated by the client (status code 101)

- WebSocket servers can share a port with HTTP servers

- WebSocket protocol is simple, co-exists peacefully with HTTP

- `ws`: a **Node.js WebSocket library**


Untitled-1 — Web-Teaching
EXPLORER
OPEN EDITORS
Untitled-1
WEB-TEACHING
.vscode
demo-code
balloons-game
node_modules
public
routes
tests
views
.eslintrc.js
app.js
game.js
package-lock.json
package.json
statTracker.js
balloons-wireframes
minimum-example-ws
node-ajax-ex
node-component-ex
node-cookies-ex
node-ejs-ex
node-express-ex
node-file-watching-ex
node-sessions-ex
node-tcp-ex
node-url-routing-ex
node-web-ex
node-websocket-ex
README.md
img
slides
.gitignore
.travis.yml
Assignment-1.md
Assignment-2.md
Assignment-3.md
OUTLINE
Untitled-1
1
Demo time: example 7!
Hello World of WebSockets:
The client sends a 'Hello' to the server, the server replies.
master
0 0
Ln 1, Col 1
Spaces: 4
UTF-8
LF

# WebSockets: on the client

```
<!DOCTYPE html>
<html>
  <head>
    <title>WebSocket test</title>
  </head>
  <body>
    <main>Status: <span id="hello"></span></main>

    <script>
      var socket = new WebSocket("ws://localhost:3000");
      socket.onmessage = function(event) {
        document.getElementById("hello").innerHTML = event.data;
      };

      socket.onopen = function() {
        socket.send("Hello from the client!");
        document.getElementById("hello").innerHTML =
          "Sending a first message to the server ...";
      };
    </script>
  </body>
</html>
```

Initiate WebSocket connection (scheme ws!)

A received message should be displayed

Message sent from client to server once the connection is open

# WebSockets: on the server

```
var express = require("express");
var http = require("http");
var websocket = require("ws");

var port = process.argv[2];
var app = express();

app.use("/", function(req, res) {
  res.sendFile("client/index.html", { root: "./" });
});
```

```
var server = http.createServer(app);

const wss = new websocket.Server({ server });

wss.on("connection", function(ws) {
  setTimeout(function() {
    console.log("Connection state: " + ws.readyState);
    ws.send("Thanks for the message. --Your server.");
    ws.close();
    console.log("Connection state: " + ws.readyState);
  }, 2000);

  ws.on("message", function incoming(message) {
    console.log("[LOG] " + message);
  });
});

server.listen(port);
```

WebSocket server object

We define what happens when a connection is established

Callback for the message event

# WebSockets: on the server

The WebSocket protocol as described in RFC 6455 has four event types:

- `open` : this event fires once a connection request has been made and the handshake was successful; messages can be exchanged now;
- `message` : this event fires when a message is received;
- `error` : something failed;
- `close` : this event fires when the connection closes; it also fires after an `onerror` event;

```
  res.sendFile("client/index.html", { root: "./" });
});

var server = http.createServer(app);

const wss = new websocket.Server({ server });

wss.on("connection", function(ws) {
  setTimeout(function() {
    console.log("Connection state: " + ws.readyState);
    ws.send("Thanks for the message. --Your server.");
    ws.close();
    console.log("Connection state: " + ws.readyState);
  }, 2000);

  ws.on("message", function incoming(message) {
    console.log("[LOG] " + message);
  });
});

server.listen(port);
```

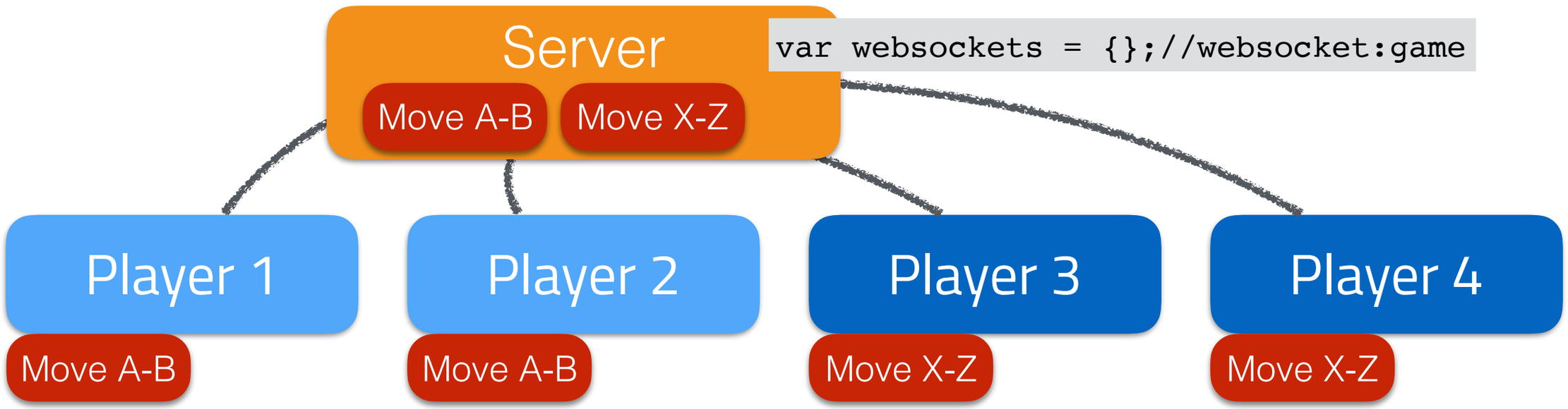
WebSocket server object

We define what happens when a connection is established

Callback for the message event

# WebSockets for multi-player games

- Every player establishes a WebSocket conn. to the server
- The server **tracks** the game each player belongs to:




```
var game = function(gameID) {
  this.playerA = null;
  this.playerB = null;
  this.id = gameID;
  this.wordToGuess = null;
  this.gameState = "0 JOINT";
};
```

# WebSockets for multi-player games

Once a client establishes a WebSocket connection, the server-side script has several tasks:

- Determine **which game** to add the player to
- Inform the player about the **game status**
- **Request information** from the player if necessary

# WebSockets for multi-player games

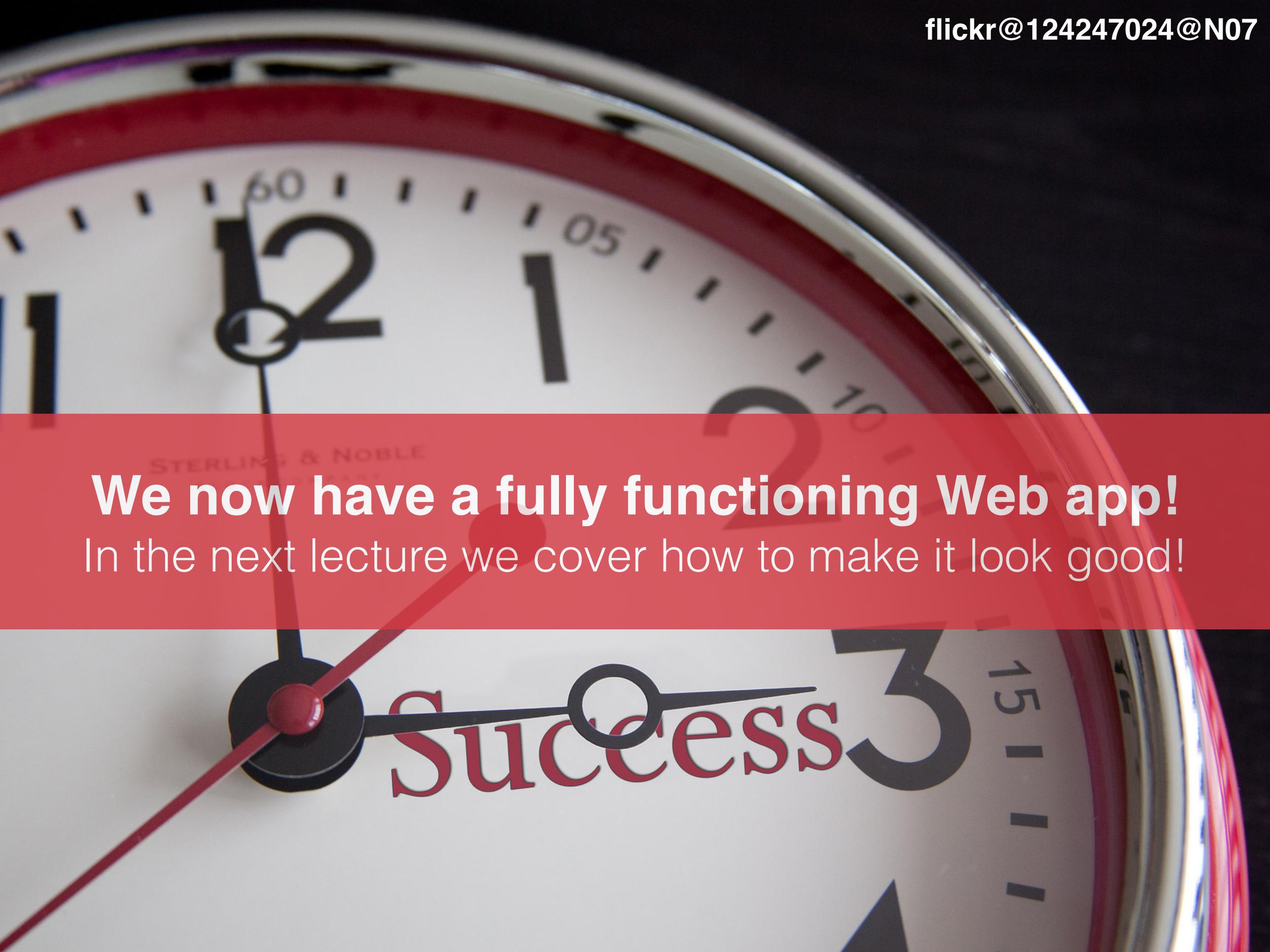
Once a client establishes a WebSocket connection, the server-side script has several tasks:

```
var currentGame = new Game(gameStatus.gamesInitialized++);
var connectionID = 0;//each websocket receives a unique ID

wss.on("connection", function connection(ws) {

    /*
     * two-player game: every two players are added to the same game
     */
    let con = ws;
    con.id = connectionID++;
    let playerType = currentGame.addPlayer(con);
    websockets[con.id] = currentGame;

    /*
     * inform the client about its assigned player type
     */
    con.send((playerType == "A") ? messages.S_PLAYER_A : messages.S_PLAYER_B);

    ...
}
```

unique identifier per WebSocket connection

add player to latest game

inform player




**We now have a fully functioning Web app!**

In the next lecture we cover how to make it look good!

- Read **chapter 3** (CSS) of the web course book!

- Work through **Assignment 5** (the most difficult one - the bulk of the app is created).

flickr@courtneyrian