Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44000 Internet Applications

Week 3 Lecture 1

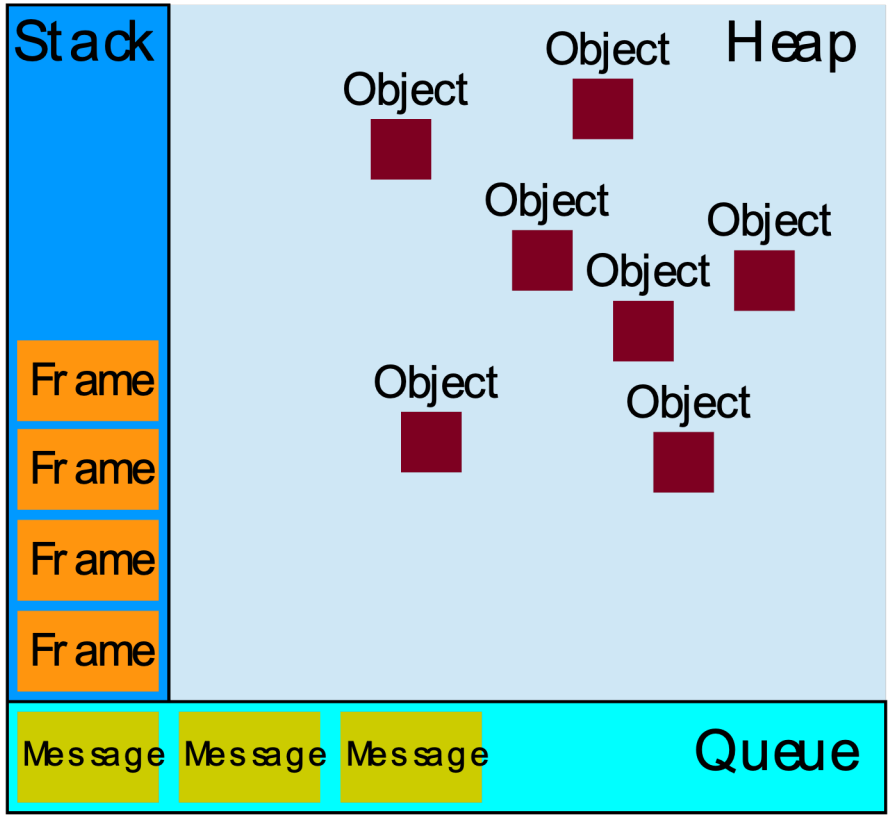
Conor Sheedy

Event Loop

JavaScript runtime model;

The Event Loop is central

- responsible for executing the code
- collecting and processing events
- executing queued sub-tasks
- different from other languages like C and Java



Event Loop

Execution context stack

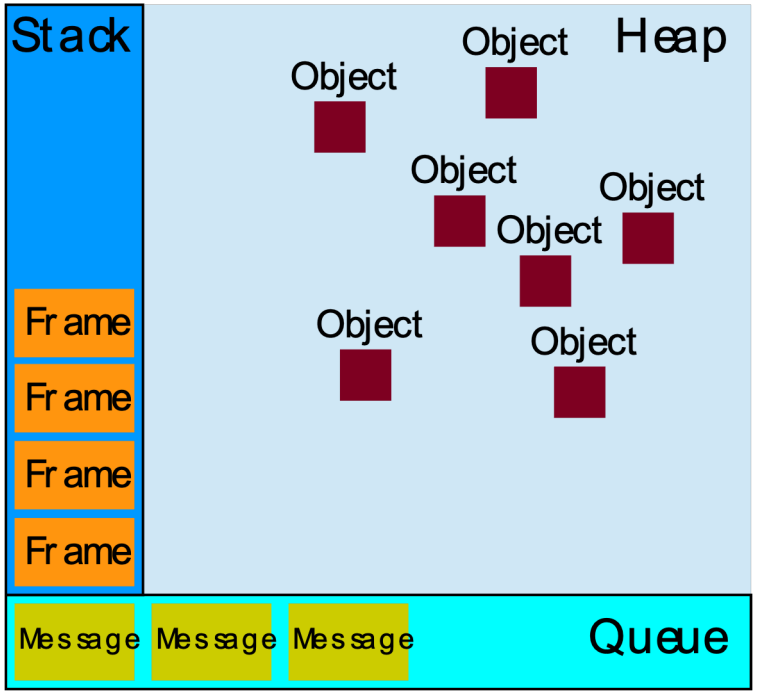
- Function calls form a stack of frames
  - stores local variables, references to functions and objects
  - One stack = single threaded

Heap

- Objects, function definitions, arrays
- Removed by Garbage Collector

Queue

- a list of messages to be processed
- Each message has an associated function
- that gets called to handle the message



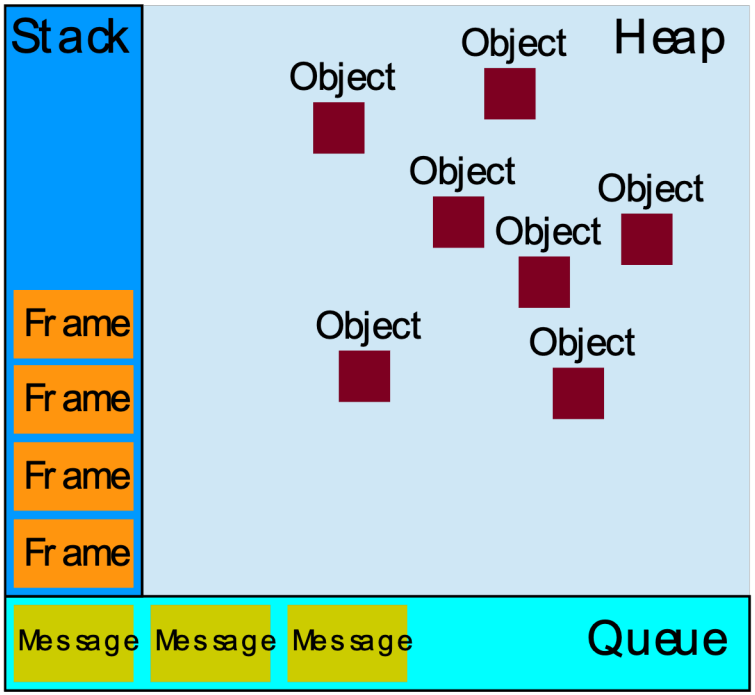
Event Loop

Browser or Web APIs

- expose data from the browser and surrounding computer environment
- not part of the JavaScript Engine
  - Part of JavaScript Runtime Environment, browser …
  - setTimeout(), DOM API, Jquery, fetch ..
  - Entered in Event Table and Waits for an event
    - Asynchronous methods

Event loop

- When the stack is empty
- the runtime starts handling the messages on the queue
- corresponding function is called with the message as an input parameter
- creates a new stack frame for that function's use




```
function printStatement() {
  console.log("Will the function delayed for 0 milliseconds be first")
}

setTimeout(printStatement, 0)

console.log("Or, will I be executed first?")
```

Event Loop

Features

- Each message is processed completely before any other message is processed
  - No pre-emption
  - Write short handler functions
  - Events need “listeners” to enter the queue

Adding Listeners to Events

- addEventListener(), 2 parameters
  - Event
    - ‘click’, ‘mouseover’, ‘dblclick’
  - Handler function

```
<button>Change color</button>
<script>
   const btn = document.querySelector('button');

   function random(number) {
   return Math.floor(Math.random() * (number+1));
   }

   btn.addEventListener('click', () => {
   const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
   document.body.style.backgroundColor = rndCol;
   });
</script>
```

True Parallelism in JavaScript

Web Workers

- Have their own thread
- Have their own stack, heap, and message queue
- run scripts in background threads
- Two distinct runtimes can only communicate through sending messages via the ‘postMessage’ method
- Considered too heavyweight for most client side use cases

Asynchronous Functions and Await

- New in ES2017
- Not yet used as widely as callbacks and promises
  - So you still have to understand them
- Offers a big improvement in readability
  - Many large projects are migrating their codebase to use async … await
- **.then** clauses queue up a series of event handlers on a promise that are executed serially
  - Chaining .then clauses lead to quite a readable codebase
- Create an Asynchronous Function
  - Using the **async** keyword
- you can **await** the fulfilment one promise after the other
  - Using the **await** keyword
  - Only possible inside an asynchronous function

```
async function doSomeStuff() {
  let result1, result2, result3;

  result1 = await promise1;
  if (result1 == “blue”)
      result2 = await promise2
  else
      result2= await promise3
}

doSomeStuff(); // kick off the async function
DoSomeOtherStuff();
```

Asynchronous Functions and Await

- While ***awaiting*** you can do minor processing in the middle
- The async function runs until it blocks and then returns a promise
  - See e.g.
  - The function has a logical sequence
- Reject results from promises can be handled as exceptions using a try…catch clause
- An Async function ALWAYS returns a promise

  - Let's analyse two examples contrasting the two styles "*promises with .then*" and "*async await*"

```
async function doSomeStuff() {
  let result1, result2, result3;

  result1 = await promise1;
  if (result1 == "blue")
      result2 = await promise2
  else
      result2= await promise3
}

doSomeStuff(); // kick off the async function
DoSomeOtherStuff();
```

Promise Example

```
// trying out promises

"use strict";
function callmebackin(s,subject)
{ return new Promise ( (resolve, reject) =>
{ setTimeout( () =>
{resolve("Returned call subject:"+subject)},
s*1000)
})
}

console.log("First I call you about a dog")
let p1 = callmebackin (5,"about a dog")
let p2 = callmebackin(3,"about another dog")
p1.then( (subject) => console.log("Got called back: "+subject))
p2.then( (subject) => console.log("you called me back:"+subject))
```

Question

What will be output to the console?

When? **Let's run it to find out.**

Example explanation

- The function 'callmebackin' has two arguments, s for a number of seconds and subject
- It returns a new promise
- There are 'resolve' and a 'reject' callbacks
- And an executor function
- Which sets a timeout for s*1000 milliseconds
- When the timeout expires it will call the resolve routine
- With the argument: "Returned call subject:"+subject

- So the objective of the 'callmebackin' function is
- To return a promise
- Which will resolve in s seconds
- And will resolve to the value : "Returned call subject:"+subject

Async… Await Example

```
// the same function as in the last example

"use strict";
function callmebackin(s,subject)
{ return new Promise ( (resolve, reject) =>
{ setTimeout( () =>
{resolve("Returned call subject:"+subject)},
s*1000)
})
}

// trying out Async Await
async function waitaround() {
let topic1 = await callmebackin(5,"about a dog");
console.log("Got called back:"+topic1)
let topic2 = await callmebackin(3,"about another dog");
console.log("you called me back:"+topic2) }

waitaround()
console.log("Hoping to get some calls")
```

Example explanation

- We use the same 'callmebackin' function as the last time
  - Which returns a promise
- This function is called twice within the asynchronous function 'waitaround'
- It produces the same result on execution but is more readable
  - Let's execute it

Order of presentation for next stage in course

Server-Side

- Using built-in http module
  - request, response pattern
  - This is the basic web server module in node.js
- The Dominant Framework which is **Express**
  - Development Pattern for express

API's

- RESTful interface principles
  - We will use express to develop API services

Client Side

- Vue Framework
  - mention Angular/React

A Simple web-server using Node's http module

http

- Node's built-in module
  - You don't have to use npm to install it
- implements a web-server

http.createServer

- Creates a server
- takes a callback with:
  - Request
    - all info about the http request that came in to the server
  - Response
    - an object that deals with issuing the response to the http request

```
// simple webserver using http built-in module
var http = require("http");

function requestHandler(request,response){
console.log("In comes a request to: "+request.url);
response.end("Hello from your new web server");
}
var server = http.createServer(requestHandler);
server.listen(3000);
```

- Full docs on: https://nodejs.org/api/http.html

A Simple web-server using Node's http module

In our example code

http.createServer

- Creates a server
  - Which will listen on port 3000
- The server object takes a callback
  - requestHandler
    - request.url
      - Gets the url mentioned in the request
    - response.end
      - Hands control back to the web server

```
// simple webserver using http built-in module
var http = require("http");

function requestHandler(request,response){
console.log("In comes a request to: "+request.url);
response.end("Hello from your new web server");
}
var server = http.createServer(requestHandler);
server.listen(3000);
```

Let's run this using node.js and check

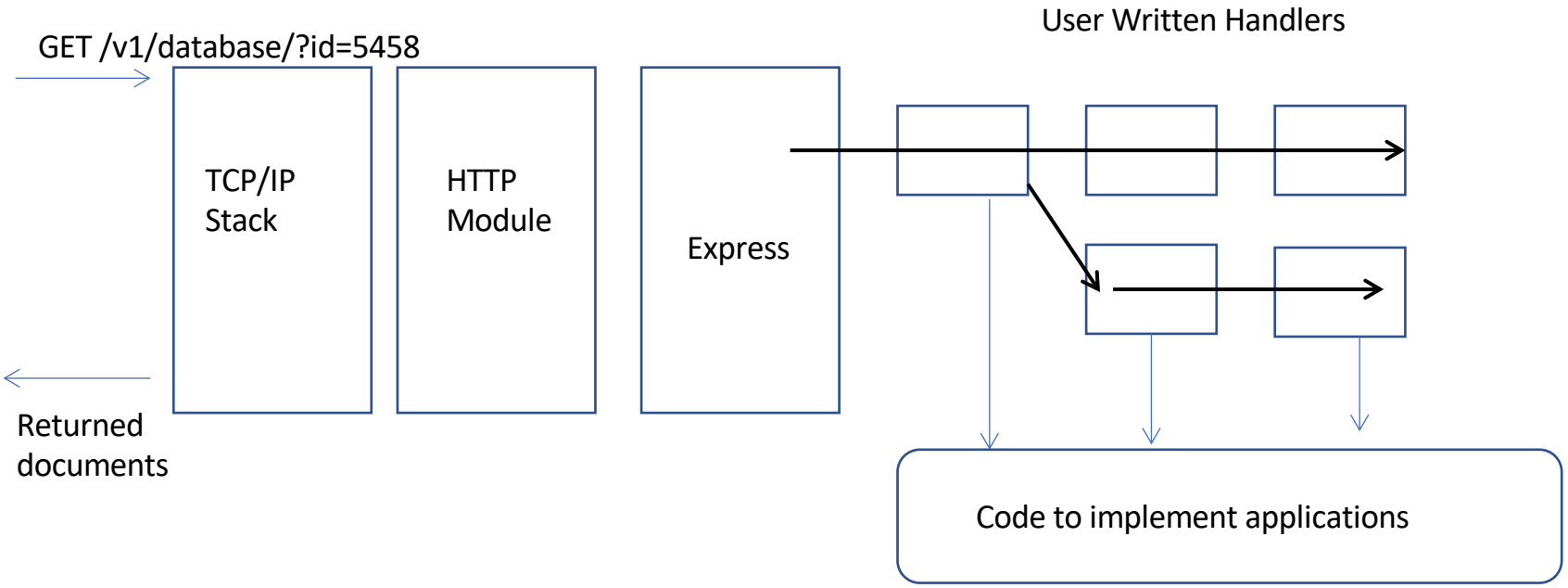
http://localhost:3000/

The Express Framework Building on Top of this built in module

Anyone can write a library that builds on this; publish it using NPM and announce it to the world.

- **Libraries that become popular become standard industry-wide**
- **until something else comes along**
- **or someone builds on top of it**
- **Express was developed by TJ Holowaychuk in May 2010**
- **In Jan 2016 it was placed under stewardship of the Node.js foundation incubator**

Express Structure



Handler Structure

Express allows handlers to be installed in a queue using primitives like

- .use, .get, .post

Each handler takes a function (request, response, next)

- Request has all the information that comes with the request
- Response is used to construct and send responses
- Next is used to pass control on to the next handler in the chain

```
"use strict";

const express = require('express')
const app = express()
const port = 4000

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```