Node.JS

Universitat Politecnica de Catalunya (UPC)

Outline

Outline

¿Qué es Node.js? I

- **Node.js** (a partir de ahora Node) es un entorno JavaScript de lado del servidor.
- Utiliza el motor V8 JavaScript de Google para interpretar y ejecutar el código.
- Soporta protocolos TCP, DNS y HTTP.
- Uno de los principales objetivos de Node es proporcionar una manera fácil para construir aplicaciones de red rápidas y escalables.
- Utiliza un modelo orientado a eventos sin bloqueo (non-blocking), que hace que sea ligero y eficiente.
- Ideal para aplicaciones en tiempo real con un gran volumen de datos.

¿Qué es Node.js? II

- Node es de código abierto, puede ser ejecutado en Linux, OS X y Microsfot Windows.
- Además, node viene con muchos módulos muy útiles, de manera que no hay que escribir todo desde cero.
- En conclusión, Node en realidad son dos cosas:
  1. Un entorno de ejecución.
  2. Una librería.

```
1 | Node.js = Runtime Environment + JavaScript Library
```

Características I

- Las siguientes son algunas de las características más importantes que están haciendo de Node.js la primera opción de los arquitectos de software.
- **Asíncrono y controlado por eventos**:
  - Todas las APIs de la librería de Node son **Asíncronas** y **No bloqueantes**.
  - Esto significa que un servidor basado en Node nunca va a esperar a que termine una peticion para devolver los datos o para pasar a la siguiente petición.
  - El servidor puede trasladarse a la siguiente petición (cuando sea llamada) y un mecanismo de notificación de eventos de Node ayuda al servidor a obtener la respuesta de la petición anterior mediante un **Callback**.

Características II

- **Muy rápido:**
  - Siendo construido sobre el motor V8 JavaScript de Google Chrome, Node es muy rápido en la ejecución del código.
- **Con un único Thread pero altamente escalable:**
  - Node utiliza un único modelo de Thread con 'Event looping'.
  - Los mecanismos de eventos ayudan al servidor a responder de forma no bloqueante y hacen que el servidor sea altamente escalable.
  - El mismo Thread puede servir un gran numero de peticiones que los servidores tradicionales como Apache HTTP Server.
- **Sin buffer:**
  - Las aplicaciones de Node nunca tienen los datos en un buffer.
  - Estas aplicaciones simplemente sacan los datos en fragmentos.

Dónde usar Node?

- A continuación se enumeran las áreas en las que Node está demostrando ser tan potente y útil:
  - Aplicaciones con E/S de datos.
  - Aplicaciones con datos en Streaming.
  - Aplicaciones en tiempo real con un gran volumen de datos.
  - Aplicaciones basadas en APIs JSON.
  - Aplicaciones de una sola página.

Callbacks I

- Node hace un uso intensivo de los **Callbacks**, por esa razón todas las Apis de Node lo soportan.
- Una función de Callback es llamada al completar una tarea determinada.
- Por ejemplo, si tenemos una función que leé un archivo, podemos comenzar a leerlo y volver al entorno de ejecución para que de inmediato sea ejecutada la próxima instrucción.
- Una vez el achivo se ha completado se llamará a una función de Callback, cuando se pasa la función el contenido del archivo se pasa como parámetro de dicha función .

Callbacks II

- De esta forma no hay bloqueo o espera para leer el archivo.
- Esto hace que Node sea altamente escalable, ya que puede procesar un número elevado de solicitudes sin esperar a ninguna función para devolver el resultado.

Blocking y Non-Blocking Code I

- Estos dos ejemplos explican claramente el concepto de bloqueo y no bloqueo de llamadas que hemos hablado anteriormente.

- **Blocking Code:**
- Vamos a suponer que tenemos un archivo llamado file.txt en nuestro directorio con el texto 'Hello World!'.
- A continuación, creamos un archivo llamado mainBC.js con el siguiente código:

```
1  var fs = require("fs");
2  var dataBC = fs.readFileSync('file.txt');
3
4  console.log(dataBC.toString());
5  console.log("Program Ended");
```

Blocking y Non-Blocking Code II

- Ahora, si ejecutamos nuestro mainBC.js mediante:

```
host$ node mainBC.js
```

- observaremos lo siguiente:

```
Hello world!
Program Ended
```

- En este primer ejemplo, el programa empieza a leer el archivo, una vez el archivo es leído procede al final del programa y muestra 'Hello world'.
- En otras palabras, un programa con bloqueo se ejecuta de forma secuencial y desde el punto de vista de la programacioń es más fácil programar la lógica.

Blocking y Non-Blocking Code III

- **Non-Blocking Code:**
- Para este caso crearemos otro archivo llamado mainNBC.js con el siguiente código:

```
1  var fs = require("fs");
2  //we use the same file.txt
3  fs.readFile('file.txt', function (err, dataNBC) {
4      if (err) return console.error(err);
5      console.log(dataNBC.toString());
6  });
7
8  console.log("Program Ended");
```

- Ejecutamos nuestro mainNBC.js de la misma manera que lo hicimos anteriormente y obtenemos

```
1  Program Ended
2  Hello world!
```

Blocking y Non-Blocking Code IV

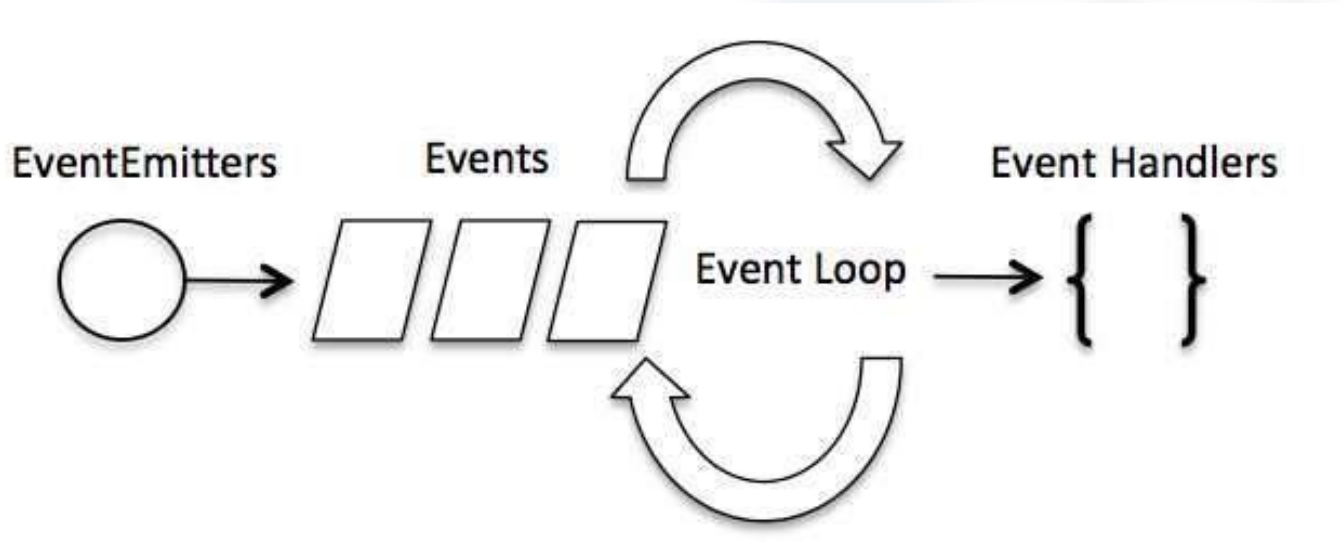
- En este caso, el programa no espera mientras está leyendo el archivo, si nos fijamos, primeramente imprime 'Program Ended' mientras sigue leyendo el archivo, luego, una vez que el archivo es leído muestra por pantalla 'Hello World!'.
- Los programas con 'No bloqueo' se ejecutan de forma **no** secuencial.

Programación orientada a eventos I

- Node soporta la concurrencia mediante los **eventos** y **CallBacks**, pese a que utiliza un único Thread.
- Dicho Thread mantiene un ciclo de eventos(event loop) y cada vez que una tarea se completa, se dispara el evento correspondiente que viene indicado por la función 'event listener'.
- Node utiliza eventos en gran medida y esta una de las razones por las cuales es bastante rápido en comparación con otras tecnologías similares.
- Al momento de iniciar el servidor, Node simplemente inicia sus variables, declara funciones y luego sencillamente espera a que se produzca un evento.
- Técnicamente Node utiliza el patrón 'Observer'.

Programación orientada a eventos II

- En una aplicación orientada a eventos, en general hay un bucle principal que escucha los eventos, luego se activa una función de callback cuando se detecta uno de esos eventos.



Programación orientada a eventos III

- Aunque los eventos parezcan similares a los callbacks.
- La diferencia radica en el hecho de que las funciones de callback son llamadas cuando una función asíncrona devuelve su resultado, en cambio los eventos trabajan con el patrón 'Observer'.
- Las funciones que escuchan a los eventos actúan como observadores, siempre que un evento se active la función de escucha inicia la ejecución.
- Node incorpora múltiples eventos, estos están disponibles a través de los módulos y la clase **EventEmitter** que se utiliza para enlazar eventos y eventos de escucha, del siguiente modo:

```
1  // Import events module
2  var events = require('events');
3  // Create an eventEmitter object
4  var eventEmitter = new events.EventEmitter();
```

Programación orientada a eventos IV

- A continuación se muestra la sintaxis para enlazar el event handler con un evento:

```
1 // Bind event and even handler as follows
2 eventEmitter.on('eventName', eventHandler);
```

- Finalmente, podemos disparar un evento de la siguiente manera:

```
1 // Fire an event
2 eventEmitter.emit('eventName');
```

Programación orientada a eventos V

- Ejemplo, Crear un archivo JS llamado main.js con el siguiente código:

```
1  // Import events module
2  var events = require('events');
3  // Create an eventEmitter object
4  var eventEmitter = new events.EventEmitter();
5
6  // Create an event handler as follows
7  var connectHandler = function connected() {
8     console.log('connection succesful.');
9
10    // Fire the data_received event
11    eventEmitter.emit('data_received');
12 }
13
14 // Bind the connection event with the handler
15 eventEmitter.on('connection', connectHandler);
16
17 // Bind the data_received event with the anonymous function
18 eventEmitter.on('data_received', function(){
19    console.log('data received succesfully.');
20 });
21 // Fire the connection event
22 eventEmitter.emit('connection');
23
24 console.log("Program Ended.");
```

Programación orientada a eventos VI

- Ejecutamos el programa y obtenemos lo siguiente:

```
1  host$ node main.js
2
3  connection succesful.
4  data received succesfully.
5  Program Ended.
```

Clase EventEmitter

- La clase EventEmitter se encuentra en el módulo de eventos. Es accesible a través de la siguiente sintaxis:

```
1  //import events module
2  var events = require('events');
3  //create an eventEmitter object
4  var eventEmitter = new events.EventEmitter();
```

- Cuando una instancia EventEmitter enfrenta cuaquier error, se emite el evento "error".
- EventEmitter ofrece múltiples propiedades como **on** y **emit**. La propiedad **on** se utiliza para enlazar una función con el evento y **emit** se utiliza para disparar el evento.

¿Cómo funciona Node? I

- En una aplicación con Node, cualquier función asíncrona acepta un callback como último párametro y las funciones callbacks aceptan 'error' como primer parámetro.
- Vamos a revisar el ejemplo anterior de nuevo.
- Creamos un archivo de texto llamado file.txt con el siguiente contenido:

```
1 | First steps with node!
```

- Luego, creamos un archivo JS llamado main.js con lo siguiente:

¿Cómo funciona Node? II

```
1   var fs = require("fs");
2
3   fs.readFile('input.txt', function (err, data) {
4     if (err){
5 console.log(err.stack);
6 return;
7     }
8     console.log(data.toString());
9   });
10  console.log("Program Ended");
```

- Primeramente tenemos que fs.readFile() es una función asíncrona cuya finalidad es la de leer un archivo.
- Si se produce un error durante la lectura del archivo, entonces el objeto 'err' contendrá los datos correspondientes a el error , y en otro caso 'data' contendrá el contenido del archivo.

¿Cómo funciona Node? III

- Luego readFile pasa 'err' y 'data' a la función de callback después de que la operación de leer el archivo se ha completado, por lo que finalmente se imprime el siguiente contenido:

```
1 Program Ended
2 First steps with node!
```

Node Package Manager (npm) I

- Node Package Manager (npm) es un gestor de paquetes de Node, que nos permite descargar librerías y enlazarlas, normalmente viene uncluido en la instalación de Node.
- **npm** proporciona las siguientes dos principales funcionalidades:
  1. Repositorios en línea para Node.js (paquetes / módulos), que se pueden buscar en search.nodejs.org.
  2. Proporciona "Command line" para instalar los paquetes de Node.js, hacer gestión de versiones y la gestión de la dependencia de los paquetes.
- **Instalando módulos usando npm:**
- La sintaxis es sencilla al momento instalar cualquier módulo de Node.js:

```
1 | $ npm install <Module Name>
```

Node Package Manager (npm) II

- Por ejemplo, el comando siguiente es para instalar un módulo famoso de Node, el framework web llamado **express**:

```
1 | $ npm install express
```

- Ahora podemos utilizar este módulo en nuestro archivo JS de la siguiente manera:

```
1 | var express = require('express');
```

Express Framework I

- **Express** es un Framework para Node, minimalista y flexible que proporciona un conjunto sólido de características para las aplicaciones web y móviles.
- Facilita un rápido desarrollo de aplicaciones Web basadas en Node.
- Las siguientes son algunas de las características centrales del Framework Express:

1. Permite configurar middlewares para responder a las peticiones HTTP.
2. Define una tabla de enrutamiento que se utiliza para llevar a cabo diferentes acciones, basadas en el método HTTP y URL.
3. Permite representar dinámicamente páginas HTML basado en el paso de argumentos a las plantillas.

Express Framework II

- **Instalando Express:**
- En primer lugar, instalar globalmente el Framework express usando **npm**, de modo que pueda ser utilizado mediante el teminal de Node.

```
$ npm install express --save
```

- Además, se recomienda instalar los siguientes módulos junto con express:

1. **body-parser**: Es un middleware de Node para la manipulación de JSON.
2. **cookie-parser**: Analiza la cabecera Cookie y llena *req.cookies* con un objeto introducido por los nombres de las cookies.
3. **multer**: Es un middleware de Node.js para el manejo de multipart/form-data.

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

Express Framework III

- **Hello world con express**:
- A continuación crearemos una aplicación básica con expreess.
- Dicha aplicación, primeramente iniciará un servidor escuchando en el puerto *8888*.
- Luego, responderá con Hello World! para las solicitudes de la página principal.
- Para este ejemplo, crearemos un archivo JS llamado server.js con el siguiente código:

Express Framework IV

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (req, res) {
5    res.send('Hello World!');
6  })
7
8  var server = app.listen(8888, function () {
9
10   var host = server.address().address
11   var port = server.address().port
12
13   console.log("Example app listening at http://%s:%s", host, port)
14
15 })
```

- Luego, lo ejecutamos mediante el comando:

```
1 $ node server.js
```

- Primero que nada, observaremos por la consola lo siguiente:

```
1 Example app listening at http://0.0.0.0:8888
```

Express Framework V

- Luego, si abrimos cualquier navegador y nos dirigimos a *http://localhost:8888*, observaremos:

```
1 | Hello World!
```

Primeros pasos con Node

- En esta parte desarrollaremos una aplicación web desde cero.
- Es necesario conocer la sintaxis y todo lo relacionado con **Javascript**, ya que como lo hemos comentado Node está basado en aquel lenguaje de programacioń.
- Empezaremos creando un servidor HTTP básico, el cual lo iremos extendiendo poco a poco.
- Antes de continuar, es importante que tengamos instalado Node en nuestro entorno de desarrollo, lo podemos descargar desde su página web oficial *https://nodejs.org/en/*.

Outline

Servidor HTTP Básico

- Empezamos con el módulo del servidor, para ello crearemos el archivo **server.js** en el directorio raíz del proyecto con el siguiente código:

```
1 var http = require("http");
2
3 http.createServer(function(request, response) {
4   response.writeHead(200, {"Content-Type": "text/
        html"});
5   response.write("Hello world");
6   response.end();
7 }).listen(8888);
```

- Con esto conseguimos crear un servidor HTTP activo.
- Si ejecutamos el script con *node server.js* y abrimos el navegador en la siguiente url *http://localhost:8888/*, observaremos una pag. web con 'Hola mundo'.

Analizando nuestro servidor HTTP

- La primera línea **require**, requiere al módulo http que viene incluido con Node y lo hace accesible a través de la variable **http**.
- Luego, llamamos a una de las funciones que el módulo **http** ofrece: **createServer**.
- Esta función retorna un objeto que tiene un método llamado **listen**, y toma un valor numérico que indica el número de puerto en que nuestro servidor HTTP va a escuchar.

Callbacks Manejados por Eventos I

- En **JavaScript**, podemos pasar una función como un parámetro cuando llamamos a otra función, es muy importante entender esta parte ya que lo utilizaremos mucho más adelante.
- En nuestro caso, estamos paśandole a la función **createServer** una función *anónima*:

```
1  http.createServer(function(request, response) {
2    ...
3
4  }).listen(8888);
```

- Cuando llamamos al método http.createServer, no solo queremos que el servidor se quede escuando en el puerto (8888), sino que también queremos hacer algo cuando haya una petición HTTP a este servidor.
- Esto sucede

Callbacks Manejados por Eventos II

- Es decir, cada vez que nuestro servidor recibe una petición, llamamos a la función que le pasamos como parámetro.
- Este concepto es llamado un **callback**.

Callbacks Manejados por Eventos III

- Vamos a entender este concepto mediante un ejemplo, para ello vamos a modificar el servidor que hemos creado previamente,

```
1  var http = require("http");
2
3  function onRequest(request, response) {
4    console.log("Peticion Recibida.");
5    response.writeHead(200, {"Content-Type": "text/html"});
6    response.write("Hello world");
7    response.end();
8  }
9
10 http.createServer(onRequest).listen(8888);
11
12 console.log("Server running...");
```

- Cuando arrancamos la aplicación (con node server.js), inmediatanemnte observaremos en la consola de comandos 'Servidor iniciado'.

Callbacks Manejados por Eventos IV

- Luego, cada vez que hagamos una petición a nuestro servidor (abriendo http://localhost:8888) , el mensaje que veremos por la consola será 'Petición recibida'.
- Esto quiere decir que nuestro código continúa ejecutandose después de haber creado el servidor, incluso si no ha se ha realizado ninguna petición HTTP.
- Esto es JavaScript de lado del servidor asíncrono y orientado al evento con callbacks en acción.

Manipulando peticiones

- Una vez se que se ejecuta la función de callback ***onRequest()***, se pasan lo siguientes dos parámetros:
  - request.
  - response.
- Estos parámetros contienen una serie de métodos que podemos usar para responder y manejar los detalles de la petición HTTP.
- En el caso de nuestro servidor, cada vez que recibimos una petición, usamos la función **response.writeHead()** para enviar un *status HTTP 200* y un *content-type* en la cabecera de la respuesta HTTP.
- La función **response.write()** la usamos para enviar el texto 'Hola Mundo'.
- Por último, llamamos a la función **response.end()** para finalizar la respuesta.

Organización del proyecto I

- Antes de continuar, vamos hablar acerca de la organización de nuestro proyecto.
- Es muy común tener un archivo principal llamado *index.js*, el cual es usado para arrancar y utlizar el resto de módulos de la aplicación.
- Como por ejemplo, en nuestro servidor ya hemos utilizado un módulo el cual es 'http':

```
1 var http = require("http");
2 ......
3 http.createServer(...);
```

- Esto quiere decir que en algún lugar dentro de Node está el módulo llamado 'http', luego se asigna el resultado a una variable local.
- Finalmente, la variable local se convierte en un objeto que contiene todos lo métodos públicos del módulo *http*.

Organización del proyecto II

- Ahora, vamos a convertir nuestro *server.js* en un módulo.
- Para hacer esto posible modificaremos el código de nuestro servidor, agregando una función llamada *start*.
- Luego exportaremos esta función.

```
1  var http = require("http");
2
3  function start() {
4    function onRequest(request, response) {
5      console.log("Peticion Recibida.");
6      response.writeHead(200, {"Content-Type": "text/html"});
7      response.write("Hola Mundo");
8      response.end();
9    }
10
11   http.createServer(onRequest).listen(8888);
12   console.log("Servidor Iniciado.");
13 }
14 exports.start = start;
```

Organización del proyecto III

- De este modo, podemos crear nuestro propio archivo principal *index.js*, y arrancar nuestro servidor HTTP desde allí.
- Creamos nuestro archivo principal *index.js*

```
1 var server = require("./server");
2 server.start();
```

- Finalmente, podemos arrancar nuestra aplicación por medio de nuestro script principal y va hacer exactamente lo mismo.

```
1 node index.js
```

- Bien, ahora podemos poner partes de nuestra aplicación en archivos diferentes y enlazarlos a través de la creación de módulos.

Organización del proyecto IV

- Por ahora tenemos una pequeñea parte de nuestra aplicación, podemos recibir peticiones HTTP, pero de momento no sabemos que hacer con ellas.
- Lo que vamos a realizar a continuación es que nuestra aplicación se comporte de forma diferente dependiendo de la URL que pongamos en nuestro navegador.