UNIVERSITÀ DI PARMA

Tecnologie Internet

a.a. 2022/2023

UNIVERSITÀ DI PARMA

Introduction to Node

Characteristic features http://www.nodejs.org

Node is a platform built on Chrome's JavaScript runtime (**V8**) for easily building fast, scalable network applications. Node.js provides an **asynchronous API** with **non-blocking I/O** that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.
Your program can make a request for a disk or network resource while doing something else, and then, when the disk or network operation has finished, a **callback** will run that handles the result.

Node uses a library called libuv (http://libuv.org) to provide access to the operating system's non-blocking network calls.

Node brings JavaScript to the server in much the same way a browser brings JavaScript to the client.
Both Node and the browser are event-driven (single thread managing an **event queue**) and non-blocking when handling I/O.

Characteristic features

“JS in the browser” example: jQuery performing an AJAX request using XMLHttpRequest (XHR)

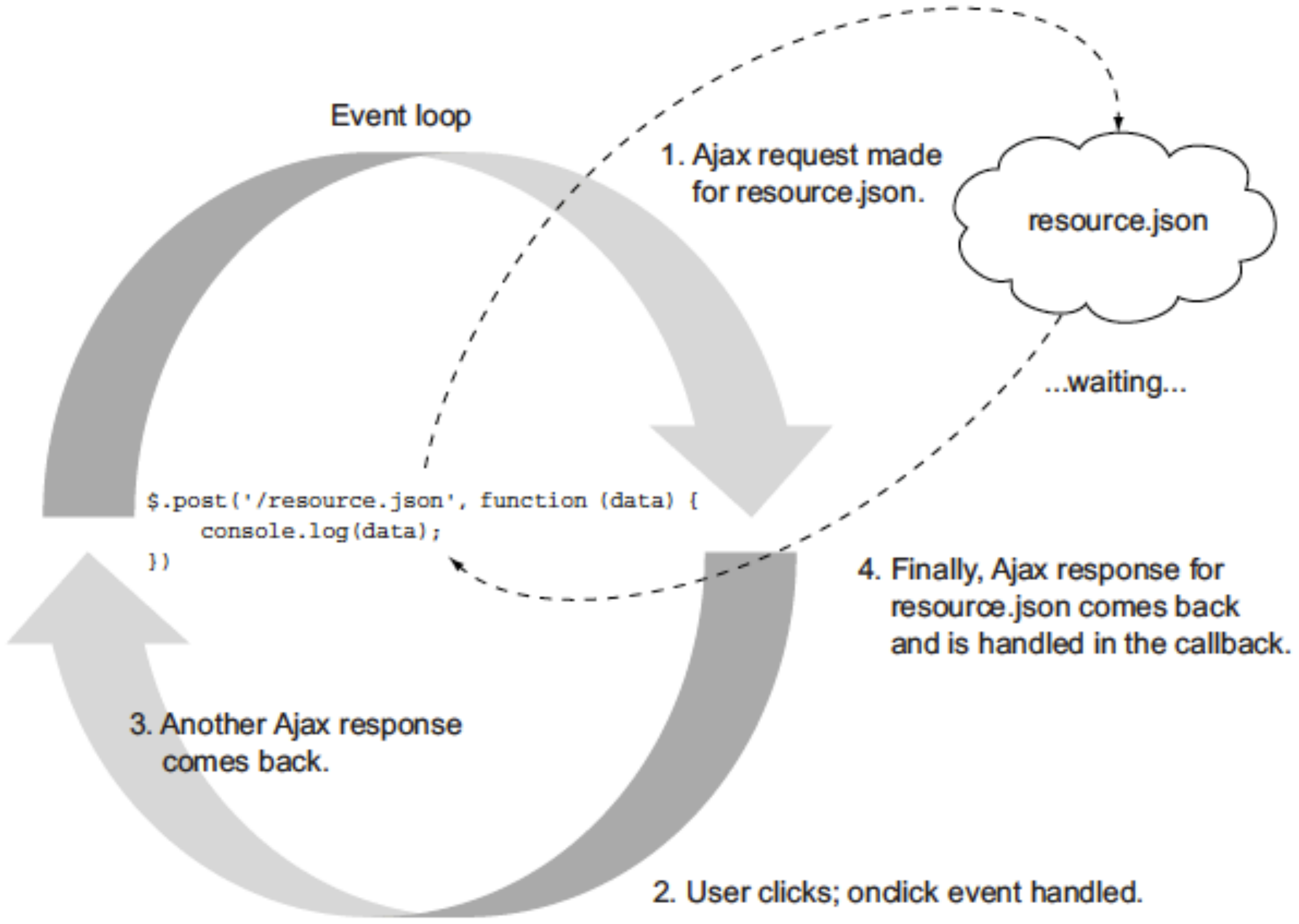
```
$.post('/resource.json', function (data) {
  console.log(data);
});
// script execution continues
```

This program performs an HTTP request for resource.json. When the response comes back, an anonymous function is called (the “callback” in this context) containing the argument data, which is the data received from that response.
I/O does not block execution! Compare with:

```
let data = $.post('/resource.json');
console.log(data);
```

Here I/O blocks execution until finished.

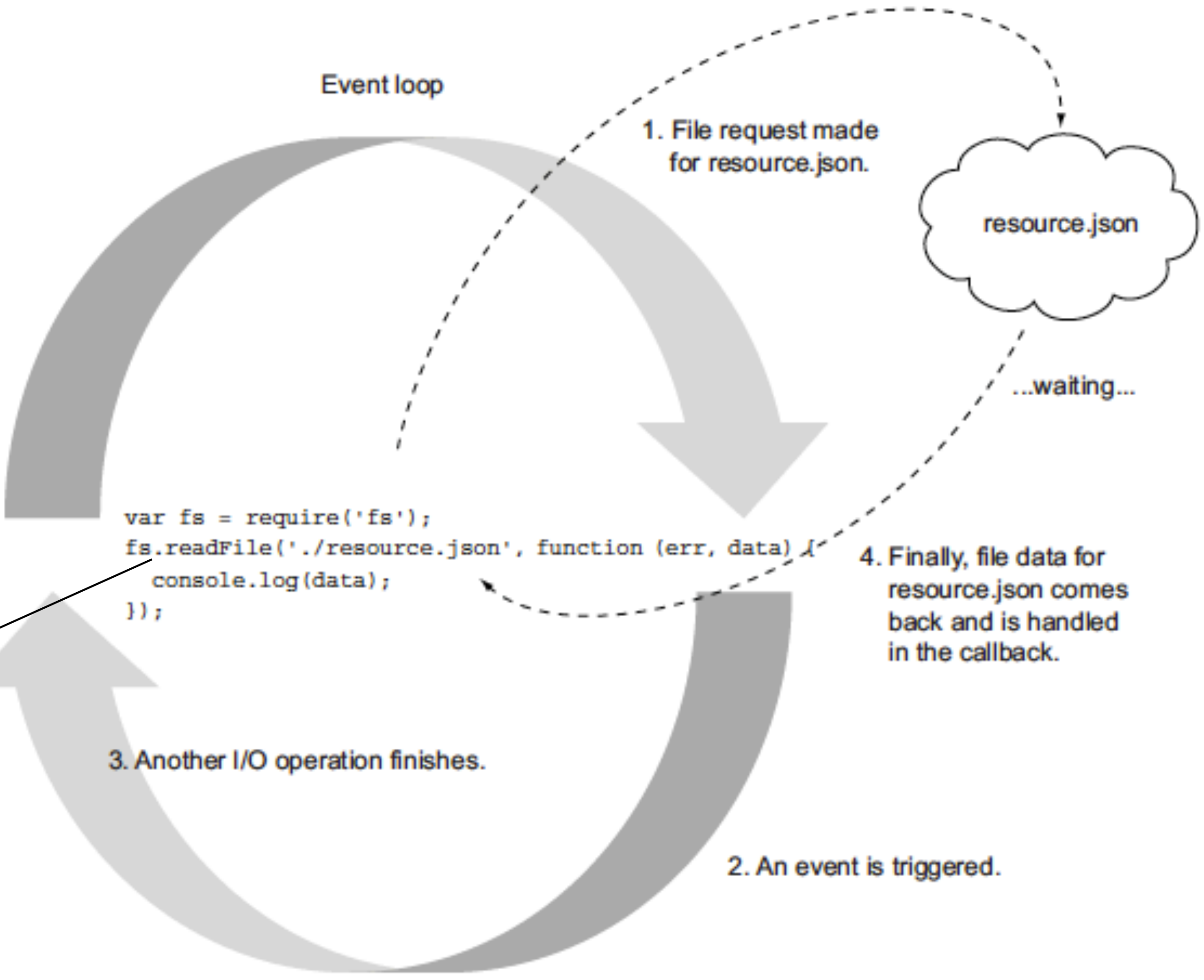
Non-blocking I/O in the browser


Event loop
1. Ajax request made for resource.json.
resource.json
...waiting...
4. Finally, Ajax response for resource.json comes back and is handled in the callback.
3. Another Ajax response comes back.
2. User clicks; onclick event handled.

```
$.post('/resource.json', function (data) {
    console.log(data);
})
```

Non-blocking I/O in Node

The difference here is that instead of making an AJAX request from the browser using jQuery, we're accessing the filesystem in Node to grab resource.json.


Event loop
1. File request made for resource.json.
resource.json
...waiting...
4. Finally, file data for resource.json comes back and is handled in the callback.
3. Another I/O operation finishes.
2. An event is triggered.

```
var fs = require('fs');
fs.readFile('./resource.json', function (err, data) {
  console.log(data);
});
```

Remember: the call to readFile doesn't actually read a file!
It schedules a file to be read and gives a reentry point when the attempt is made.

DIRT applications

Node is designed for **data-intensive real-time (DIRT)** applications.

Node itself is designed to be **responsive**, like the browser.

Node tries to **keep consistency between the browser and the server** by reimplementing common host objects, such as these:
- Timer API (for example, setTimeout)
- Console API (for example, console.log)

Node also includes a core set of modules for many types of network and file I/O. These include modules for
- HTTP, HTTPS
- filesystem (POSIX)
- DB access
- etc.

Projects, applications, companies using Node

- eBay
- Linkedin: uses Node as the server interface for it's mobile applications
- PayPal: created the http://krakenjs.com framework
- Uber
- Yahoo!
- Netflix
- Walmart
- Groupon

etc.

Installation

Download Node source code or a pre-built installer from

https://nodejs.org/en/download/

Latest stable version: **v18.12**

API reference documentation: https://nodejs.org/api/

Coding tutorials: https://www.w3schools.com/nodejs/

Example 1

the callback

```
let http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

Save the code in a file (example1.js) and run it with Node:

```
> node example1.js
```

then look for http://127.0.0.1:8124/ with your browser.

Example 2

```
let http = require('http');
let fs = require('fs');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'image/png'});
  fs.createReadStream('./image.png').pipe(res);
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

The data is read in from the file (fs.createReadStream) and is sent out (.pipe) to the client (res) as it comes in. The event loop is able to handle other events while the data is being streamed.

Save the code in a file (example2.js) and run it with Node:

```
>  node example2.js
```

then look for http://localhost:3000/ with your browser.

Example 3

example3.js: serving HTML files, CSS files and js files

```
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  console.log(request.url);
  if ((request.url == "/index") || (request.url == "/index.html")) {
          sendFileContent(response, "index.html", "text/html");
     }
     else if ((request.url == "/second") || (request.url == "/second.html")) {
     sendFileContent(response, "second.html", "text/html");
     }
     // The test() method tests for a match in a string.
     else if (/^\/[a-zA-Z0-9\/]*.css$/.test(request.url.toString())) {
          sendFileContent(response, request.url.toString().substring(1), "text/css");
     }
     else if (/^\/[a-zA-Z0-9\/]*.js$/.test(request.url.toString())){
          sendFileContent(response, request.url.toString().substring(1), "text/
javascript");
     }
     else {
          response.writeHead(200, {'Content-Type': "text/html"});
          response.write('<b>Hey there!</b><br /><br />This is the default response.
Requested URL is: ' + request.url);
          response.end();
     }
}).listen(3003);
```

Example 3

example3.js: serving HTML files, CSS files and js files

```
function sendFileContent(response, fileName, contentType){
     fs.readFile(fileName, function(err, data){
          if(err){
               response.writeHead(404);
               response.write("Not Found!");
          }
          else{
               response.writeHead(200, {'Content-Type': contentType});
               response.write(data);
          }
          response.end();
     });
}
```

```
> node example3.js
```

Look for http://localhost:3003/ with your browser
Then look for http://localhost:3003/index

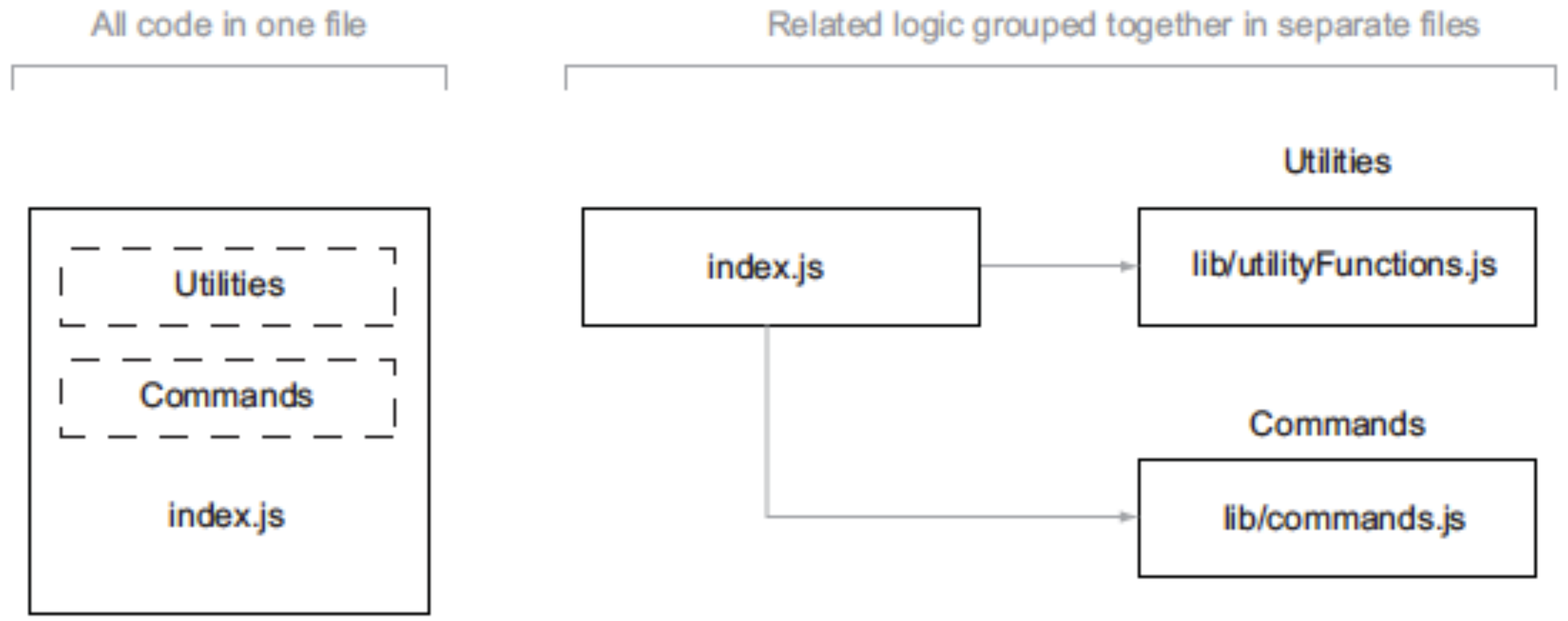
UNIVERSITÀ DI PARMA

Modularity

UNIVERSITÀ DI PARMA

Modularity

It's easier to navigate your code if you organize it using directories and separate files rather than keeping your application in one long file.



Node module system provides a powerful mechanism for organizing code.

UNIVERSITÀ DI PARMA

Modularity

Node modules allow you to select what functions and variables from the included file are exposed to the application.

If the module is returning more than one function or variable, the module can specify these by setting the properties of an object called `exports`.

If the module is returning a single function or variable, the property `module.exports` can instead be set.

The **npm (Node Package Manager)** repository is an online collection of ready-to-use Node modules.

Creating modules

Modules can either be single files or directories containing one or more files.

If a module is a directory, the file in the module directory that will be evaluated is normally named index.js.

```
let canadianDollar = 0.91;

function roundTwoDecimals(amount) {
  return Math.round(amount * 100) / 100;
}

exports.canadianToUS = function(canadian) {
  return roundTwoDecimals(canadian * canadianDollar);
}

exports.USToCanadian = function(us) {
  return roundTwoDecimals(us / canadianDollar);
}
```

save this as currency.js

canadianToUS function can be used by code requiring this module

USToCanadian function as well

Creating modules

To utilize your new module, use Node’s **`require`** function, which takes a path to the module you wish to use as an argument.

`require` is one of the few synchronous I/O operations available in Node.

Avoid using `require` in I/O-intensive parts of your application!

Any synchronous call will block Node from doing anything until the call has finished. For example, if you’re running an HTTP server, you would take a performance hit if you used `require` on each incoming request. This is typically why `require` and other synchronous operations are used only when the application initially loads.

Creating modules

Path uses ./ to indicate that module exists within same directory as application script.

The .js in currency.js can be omitted.

```
let currency = require('./currency');

console.log('50 Canadian dollars equals this amount of US
dollars:');
console.log(currency.canadianToUS(50));

console.log('30 US dollars equals this amount of Canadian
dollars:');
console.log(currency.USToCanadian(30));
```

Fine-tuning module creation using `module.exports`

The currency converter module created earlier could be redone to return a single Currency constructor function rather than an object containing functions.

```
let Currency = function(canadianDollar) {
  this.canadianDollar = canadianDollar;
}

Currency.prototype.roundTwoDecimals = function(amount) {
  return Math.round(amount * 100) / 100;
}

Currency.prototype.canadianToUS = function(canadian) {
  return this.roundTwoDecimals(canadian * this.canadianDollar);
}

Currency.prototype.USToCanadian = function(us) {
  return this.roundTwoDecimals(us / this.canadianDollar);
}

module.exports = Currency;
```

Fine-tuning module creation using `module.exports`

This is how the Currency constructor function must be used:

```
let Currency = require('./currency');

let canadianDollar = 0.91;

let currency = new Currency(canadianDollar);

console.log(currency.canadianToUS(50));
```

Packaging modules

If a module is a directory, the file in the module directory that will be evaluated must be named index.js, unless specified otherwise by a file in the module directory named **package.json**.

To specify an alternative to index.js, the package.json file must contain JavaScript Object Notation (JSON) data defining an object with a key named main that specifies the path, within the module directory, to the main file.

```
{
  "main": "./currency.js"
}
```

Asynchronous programming

Asynchronous programming techniques

Callbacks generally define logic to be executed at the completion of a given asynchronous task.
If you perform a database query, for example, you can specify a callback to determine what to do with the query results. The callback may display the database results, do a calculation based on the results, or call another function using the query results as an argument.

Event emitters fire events and include the ability to handle those events when triggered.

Callbacks

A **callback** is a function, passed as an argument to an asynchronous function, that describes what to do after the asynchronous operation has completed.

Let’s make a simple HTTP server that does the following:

- Pull the titles of recent posts stored as a JSON file asynchronously
- Pull a basic HTML template asynchronously
- Assemble an HTML page containing the titles
- Send the HTML page to the user

Callbacks

JSON file (titles.json):

```
[
  “Kazakhstan is a huge country... what goes on there?”,
  “This weather is making me craaazy”,
  “My neighbor sort of howls at night”
]
```

HTML template file (template.html):

```
<!doctype html>
<html>
  <head></head>
  <body>
    <h1>Latest Posts</h1>
    <ul><li>%</li></ul>
  </body>
</html>
```

Callbacks

HTTP server (callback_example.js):

```
let http = require('http');
let fs = require('fs');

let server = http.createServer(function (req, res)
  getTitles(res);
}).listen(8000, "127.0.0.1");

function getTitles(res) {
  fs.readFile('./titles.json', function (err, data) {
    if (err)
      return hadError(err, res);
    getTemplate(JSON.parse(data.toString()), res);
  })
}

...
```

Create HTTP server and use callback to define response logic.

Read data from JSON file and use callback to define what to do with such data.

Most Node built-in modules use callbacks with two arguments: the first argument is for an error, should one occur, and the second argument is for the results. The error argument is often abbreviated as er or err.

Callbacks

HTTP server (callback_example.js):

```
function getTemplate(titles, res) {
  fs.readFile('./template.html', function (err, data) {
    if (err)
      return hadError(err, res);
    formatHtml(titles, data.toString(), res);
  })
}

function formatHtml(titles, tmpl, res) {
  let html = tmpl.replace('%', titles.join('</li><li>'));
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end(html);
}

function hadError(err, res) {
  console.error(err);
  res.end('Server Error');
}
```

Read HTML file and use callback to process it (by calling a specific function).

Fill the HTML file with data, then send it to the client.

If an error occurs along the way, hadError logs error to console and responds to the client with “Server Error”.

Event emitters

Event emitters fire events and include the ability to handle those events when triggered.

Some important Node API components, such as HTTP servers, TCP servers, and streams, are implemented as event emitters. You can also create your own.

For example, a TCP socket in Node has an event called ‘data’ that’s triggered whenever new data is available on the socket. We set a listener for such an event:

```
let net = require('net');
let server = net.createServer(function(socket) {
  socket.on('data', function(data) {
    socket.write(data);
  });
});
server.listen(8888);
```

Run this echo server with: `node echo_server.js`

Then connect to the server with: `telnet 127.0.0.1 8888`

Event emitters

Now, let’s implement our own event emitter.

The following code defines a channel event emitter with a single listener that responds to someone joining the channel.

```
let events = require('events');

let channel = new events.EventEmitter();

channel.on('join', function() {
  console.log("Welcome!");
});
```

Then, to trigger a ‘join’ event use the emit function:

```
channel.emit('join');
```

Callbacks vs Event Emitters

Node uses EventEmitter internally a lot but exposes callback APIs to abstract/simplify the process.

As an example, the HTTP server emits a ‘request’ event upon receiving an HTTP request. The server itself listens for that request event to occur, provided with response logic for handling the event. This is implicit in:

```
http
  .createServer(respondToRequest)
  .listen(1337, '127.0.0.1');
```

but could be explicitly stated as:

```
let server = http.createServer();
server.on('request', respondToRequest);
server.listen(1337, '127.0.0.1');
```

Chat server example

```
let events = require('events');
let net = require('net');

let channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};

channel.on('join', function(id, client) {
  this.clients[id] = client;
  this.subscriptions[id] = function(senderId, message) {
    if (id != senderId) {
      this.clients[id].write(message);
    }
  }
  this.on('broadcast', this.subscriptions[id]);
});

...
```

Add a listener for the join event that stores a user's client object, allowing the application to send data back to the user.

Ignore data if it has been directly broadcast by the user.

Add a listener, specific to the current user, for the broadcast event.

Chat server example

```
let server = net.createServer(function (client) {
  let id = client.remoteAddress + ':' + client.remotePort;
  console.log("New client connected: " + id);

  channel.emit('join', id, client);

  client.on('data', function(data) {
    let message = id + ": " + data.toString();
    channel.emit('broadcast', id, message );
  });
});

server.listen(8888);
```

Emit a join event when a user connects to the server, specifying the user ID and client object.

Emit a channel broadcast event, specifying the user ID and message, when any user sends data.

After you have the chat server running, open a new command line and enter the following code to enter the chat: telnet 127.0.0.1 8888

If you open up a few command lines, you'll see that anything typed in one command line is echoed to the others.

Chat server example

The problem with this chat server is that when users close their connection and leave the chat room, they leave behind a listener that will attempt to write to a client that's no longer connected. This will, of course, generate an error.
To fix this issue, you need to add the listener in the following listing to the channel event emitter, and add logic to the server's close event listener to emit the channel's leave event. The leave event essentially removes the broadcast listener originally added for the client.

```
...
channel.on('leave', function(id) {
  channel.removeListener('broadcast', this.subscriptions[id]);
  channel.emit('broadcast', id, id + " has left the chat.\n");
});

let server = net.createServer(function (client) {
  ...
  client.on('close', function() {
    channel.emit('leave', id);
  });
});
server.listen(8888);
```

Chat server example

If you want to prevent a chat for some reason, but don’t want to shut down the server, you could use the removeAllListeners event emitter method to remove all listeners of a given type. The following code shows how this could be implemented for our chat server example:

```
channel.on('shutdown', function() {
  channel.emit('broadcast', '', "Chat has shut down.\n");
  channel.removeAllListeners('broadcast');
});
```

You could then add support for a chat command that would trigger the shutdown. To do so, change the listener for the data event to the following code:

```
client.on('data', function(data) {
  data = data.toString();
  if (data == "shutdown\r\n") {
    channel.emit('shutdown');
  }
  channel.emit('broadcast', id, data);
});
```

Sequencing asynchronous logic

Sequencing asynchronous logic

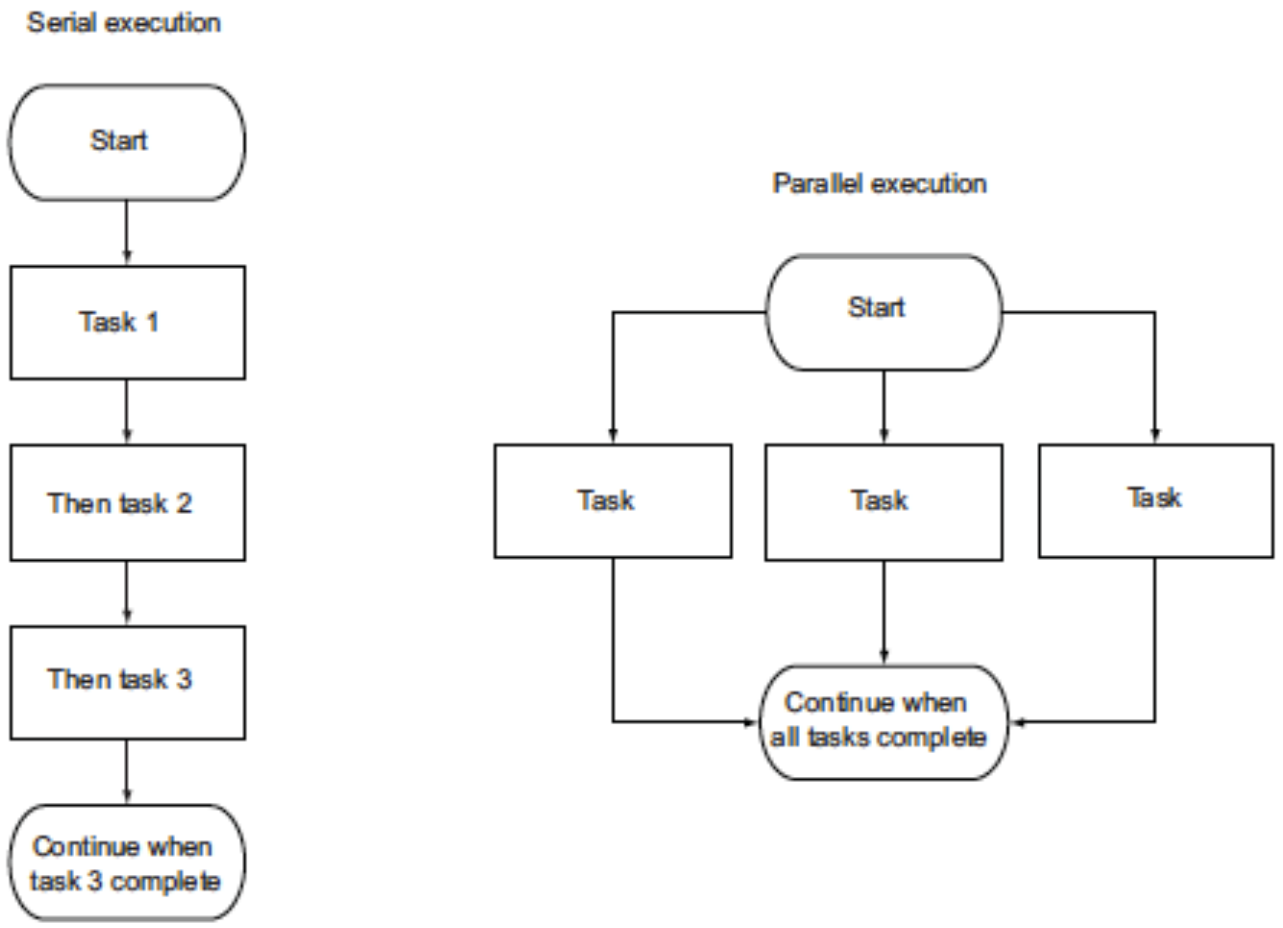
During the execution of an asynchronous program, there are some tasks that can happen any time, independent of what the rest of the program is doing, without causing problems.

But there are also some tasks, however, that should happen only before or after certain other tasks.

The concept of sequencing groups of asynchronous tasks is called **flow control** by the Node community.

Sequencing asynchronous logic

There are two types of flow control: *serial* and *parallel*.



Implementing serial flow control

We'll make a simple application that will display a single article's title and URL from a randomly chosen RSS feed.

Our example requires the use of two helper modules from the npm repository. First, open a command-line prompt, and then enter the following commands to create a directory for the example and install the helper modules:

```
mkdir random_story
cd random_story
npm install request
npm install htmlparser
```

The **request module** is a simplified HTTP client that you can use to fetch RSS data.
The **htmlparser module** has functionality that will allow you to turn raw RSS data into JavaScript data structures.

Implementing serial flow control

Next, create a file named random_story.js inside your new directory that contains the code shown in the following.

```
let fs = require('fs');
let request = require('request');
let htmlparser = require('htmlparser');
let configFilename = './rss_feeds.txt';


function checkForRSSFile () {
  fs.exists(configFilename, function(exists) {
    if (!exists)
      return next(new Error('Missing RSS file: ' + configFilename));
    next(null, configFilename);
  });
}

...
```

Task 1: Make sure file containing the list of RSS feed URLs exists.

Implementing serial flow control

```
...

function readRSSFile (configFilename) {
  fs.readFile(configFilename, function(err, feedList) {
    if (err) return next(err);
    feedList = feedList
                .toString()
                .replace(/^\s+|\s+$/g, '')
                .split("\n");
    let random = Math.floor(Math.random()*feedList.length);
    next(null, feedList[random]);
  });
}

...
```

Task 2: Read and parse file containing the feed URLs.

Convert list of feed URLs to a string and then into an array of feed URLs.

Select random feed URL from array of feed URLs.

check this with https://regex101.com/

Implementing serial flow control

```
...

function downloadRSSFeed (feedUrl) {
  request({uri: feedUrl}, function(err, res, body) {
    if (err) return next(err);
    if (res.statusCode != 200)
      return next(new Error('Abnormal response status code'))
    next(null, body);
  });
}

...
```

Task 3: Make an HTTP request and get data for the selected feed.

Implementing serial flow control

```
...

function parseRSSFeed (rss) {
  let handler = new htmlparser.RssHandler();
  let parser = new htmlparser.Parser(handler);
  parser.parseComplete(rss);
  if (!handler.dom.items.length)
    return next(new Error('No RSS items found'));
  let item = handler.dom.items.shift();
  console.log(item.title);
  console.log(item.link);
}

...
```

Task 4: Parse RSS data into array of items.

Display title and URL of the first feed item, if it exists.

Implementing serial flow control

```
...

let tasks = [ checkForRSSFile,
              readRSSFile,
              downloadRSSFeed,
              parseRSSFeed ];

function next(err, result) {
  if (err) throw err;
  let currentTask = tasks.shift();
  if (currentTask) {
    currentTask(result);
  }
}

next();
```

Add each task to be performed to an array in execution order.

A function called next executes each task. Throw exception if task encounters an error.

Execute current task.

Start serial execution tasks.

Implementing parallel flow control

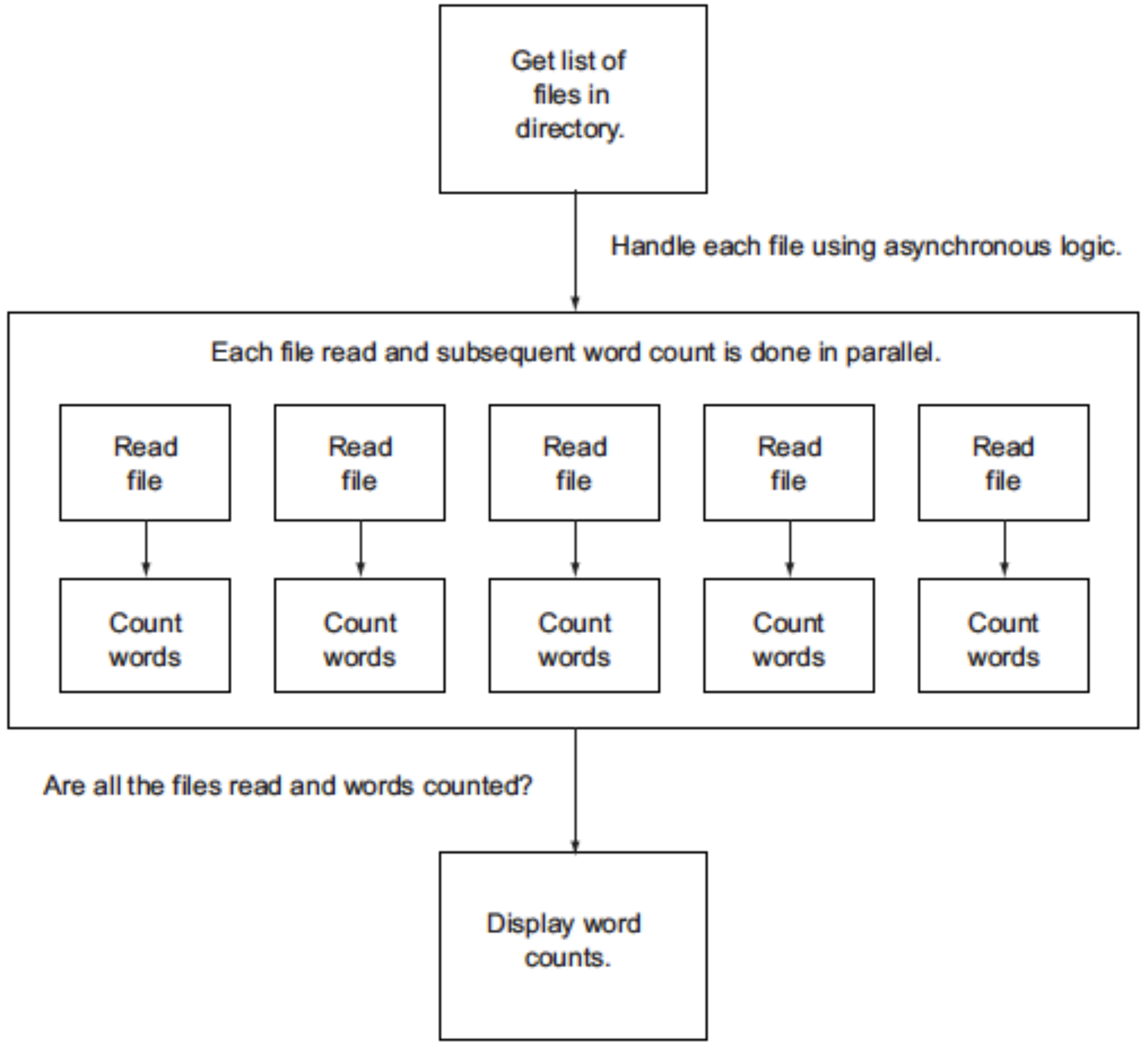
In order to execute a number of asynchronous tasks in parallel, you again need to put the tasks in an array, but this time **the order of the tasks is unimportant**.

Each task should call a handler function that will **increment a counter of completed tasks**. When all tasks are complete, the handler function should perform some subsequent logic.

For a parallel flow control example, we'll make a simple application that will read the contents of a number of text files and output the word frequencies.

Reading the contents of the text files will be done using the asynchronous readFile function, so a number of file reads could be done in parallel.

Implementing parallel flow control


Get list of files in directory.
Handle each file using asynchronous logic.
Each file read and subsequent word count is done in parallel.
Read file
Count words
Read file
Count words
Read file
Count words
Read file
Count words
Read file
Count words
Are all the files read and words counted?
Display word counts.

Implementing parallel flow control

Open a command-line prompt and enter the following commands to create two directories - one for the example, and another within that to contain the text files you want to analyze:

```
mkdir word-count
cd word-count
mkdir text
```

Next, create a file named word-count.js inside the word-count directory that contains the code that follows.

Implementing parallel flow control

```
let fs = require('fs');
let completedTasks = 0;
let tasks = [];
let wordCounts = {};
let filesDir = './text';

function checkIfComplete() {
  completedTasks++;
  if (completedTasks == tasks.length) {
    for (let index in wordCounts) {
      console.log(index + ': ' + wordCounts[index]);
    }
  }
}

...
```

When all tasks have completed, list each word used in the files and how many times it was used.

Implementing parallel flow control

```
...

function countWordsInText(text) {
  let words = text
    .toString()
    .toLowerCase()
    .split(/\W+/)
    .sort();
  for (let index in words) {
    let word = words[index];
    if (word) {
      wordCounts[word] =
        (wordCounts[word]) ? wordCounts[word] + 1 : 1;
    }
  }
}

...
```

Count word occurrences in text.

Implementing parallel flow control

```
...

// The execution starts here
fs.readdir(filesDir, function(err, files) {
  if (err) throw err;
  for(let index in files) {
    let task = (function(file) {
      return function() {
        fs.readFile(file, function(err, text) {
          if (err) throw err;
          countWordsInText(text);
          checkIfComplete();
        });
      }
    })(filesDir + '/' + files[index]);
    tasks.push(task);
  }
  for(let task in tasks) {
    tasks[task]();
  }
});
```

Get a list of the files in the ./text directory.

Define a task to handle each file. Each task includes a call to a function that will asynchronously read the file and then count the file's word usage.

Add each tasks to an array of functions to call in parallel. Then start executing the tasks.

UNIVERSITÀ DI PARMA

Node and MySQL

Node and MySQL

To access a MySQL database with Node.js, you need a MySQL driver. Here we use the "mysql" module, downloaded from NPM.

Install:

```
$ npm install mysql
```

Connecting to the database

As an example, we connect to a WordPress database:

```
let mysql = require('mysql');

let connection = mysql.createConnection(
    {
      host     : 'localhost',
      user     : 'your-username',
      password : 'your-password',
      database : 'wordpress',
    }
);

connection.connect();
```

Querying the database

```
let queryString = 'SELECT * FROM wp_posts';

connection.query(queryString, function(err, result, fields) {
    if (err) throw err;

    for (let i in result) {
        console.log('Post Titles: ', result[i].post_title);
    }
});

connection.end();
```

Query results

The query results are returned as an array of objects, each objects corresponding to a table row. An example object is shown below.

```
{ ID: '21',
  post_author: '1',
  post_date: Fri Feb 27 2009 03:44:07 GMT+0530 (India Standard Time),
  post_date_gmt: Fri Feb 27 2009 03:44:07 GMT+0530 (India Standard Time),
  post_content: 'sample content',
  post_title: 'web scrape',
  post_category: 0,
  post_excerpt: '',
  post_status: 'publish',
  comment_status: 'open',
  ping_status: 'open',
  ...
  post_mime_type: '',
  comment_count: '0' },
```

UNIVERSITÀ DI PARMA

Query results

We can access each field (`post_title` shown here) as below.

```
rows[i].post_title
```

The ‘query’ method of the connection object requires a callback function which will be executed whenever either one of the three events fires – error, result, fields. Here the callback is registered as a anonymous function.

Processing each row as it arrives

The above code with the callback defined as an anonymous function will return the results as a single data stream. However if there are a large number of rows in the table and you want to process each row as it arrives, rather then waiting to collect all the rows, you can change the code to the following.

```
let query = connection.query('SELECT * FROM wp_posts');

query.on('result', function(row) {
    console.log(row.post_title);
});

connection.end();
```

Pausing the query

Note that each row is written to the console as it arrives. If you need to process the row for some purpose before getting the next row, you will have to pause the query and resume after some processing is done.

```
query.on('result', function(row) {
    connection.pause();
    // Do some more processing on the row
    console.log(row);
    connection.resume();
});
```

Handling the ‘close’ event

Your connection to the MySQL server may close unexpectedly due to an error or you may close it explicitly. If it closes due to some error then you will need to handle that and reopen it if required. The 'close' event is fired when a connection is closed, so we need to handle that.

```
connection.on('close', function(err) {
  if (err) {
    // Oops! Unexpected closing of connection, let’s reconnect back.
    connection = mysql.createConnection(connection.config);
  } else {
    console.log('Connection closed normally.');
  }
});
```

The `connection.config` object holds the current connections details which you can use to reconnect back to the MySQL server.

UNIVERSITÀ DI PARMA

crypto Module

Why crypto

For most users, the built-in tls module and https module should more than suffice.

But if you want to implement your own security protocol, based on cryptographic primitives, crypto is very useful!

crypto provides:

- ciphers
- cryptographic hash functions
- HMAC
- signing and verification

Ciphers

Example

```
const crypto = require('crypto');
const secret = 'pprvbraumutjsrssaizxyvvlgllojzlw'
const iv = Buffer.from(crypto.randomBytes(16)); // a Buffer with 16 random bytes
const ivHex = iv.toString("hex");
const cipher = crypto.createCipheriv('aes-256-ctr', Buffer.from(secret), iv);
const encryptedPassword = Buffer.concat([
        cipher.update('mypassword'),
        cipher.final(),
    ]); // encrypted password (still Buffered, though)
const encryptedPasswordHex = encryptedPassword.toString("hex");
console.log(encryptedPasswordHex);

const decipher = crypto.createDecipheriv('aes-256-ctr', Buffer.from(secret),
Buffer.from(ivHex, "hex"));
const decryptedPassword =
Buffer.concat([decipher.update(Buffer.from(encryptedPasswordHex, "hex")),decipher.final()]);
console.log(decryptedPassword.toString());
```

Cryptographic hash functions

A hash is a fixed-length string of bits that is procedurally and deterministically generated from some arbitrary block of source data.

Properties of cryptographic hash functions:
- collision-resistant
- unidirectional

Example

```
const crypto = require('crypto');

const hashHex = crypto.createHash('sha256').update('Node is wonderful!').digest('hex');

console.log(hashHex);
```

HMAC

HMAC stands for **Hash-based Message Authentication Code**, and is a process for applying a hash algorithm to both data and a secret key that results in a single final hash.

Example

```
const crypto = require('crypto');

const hmacHex = crypto.createHmac('sha256','key').update("Node is great!").digest('hex');

console.log(hmacHex);
```

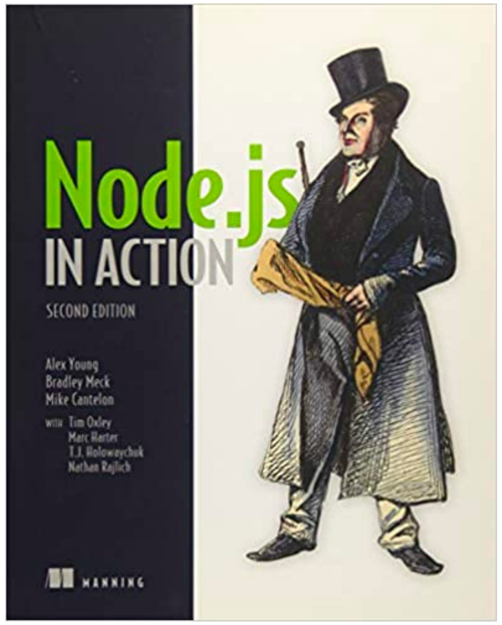
Signing and verification

Crypto has other methods used for dealing with certificates and credentials:

- `crypto.createSign`
- `crypto.createVerify`

https://nodejs.org/api/crypto.html

References



A. Young, B. Meck, M. Cantelon, T. Oxley, “Node.js in action”, 2nd ed., Manning, 2018

https://nodejs.org

https://www.w3schools.com/nodejs/nodejs_mysql.asp

https://nodejs.org/api/crypto.html