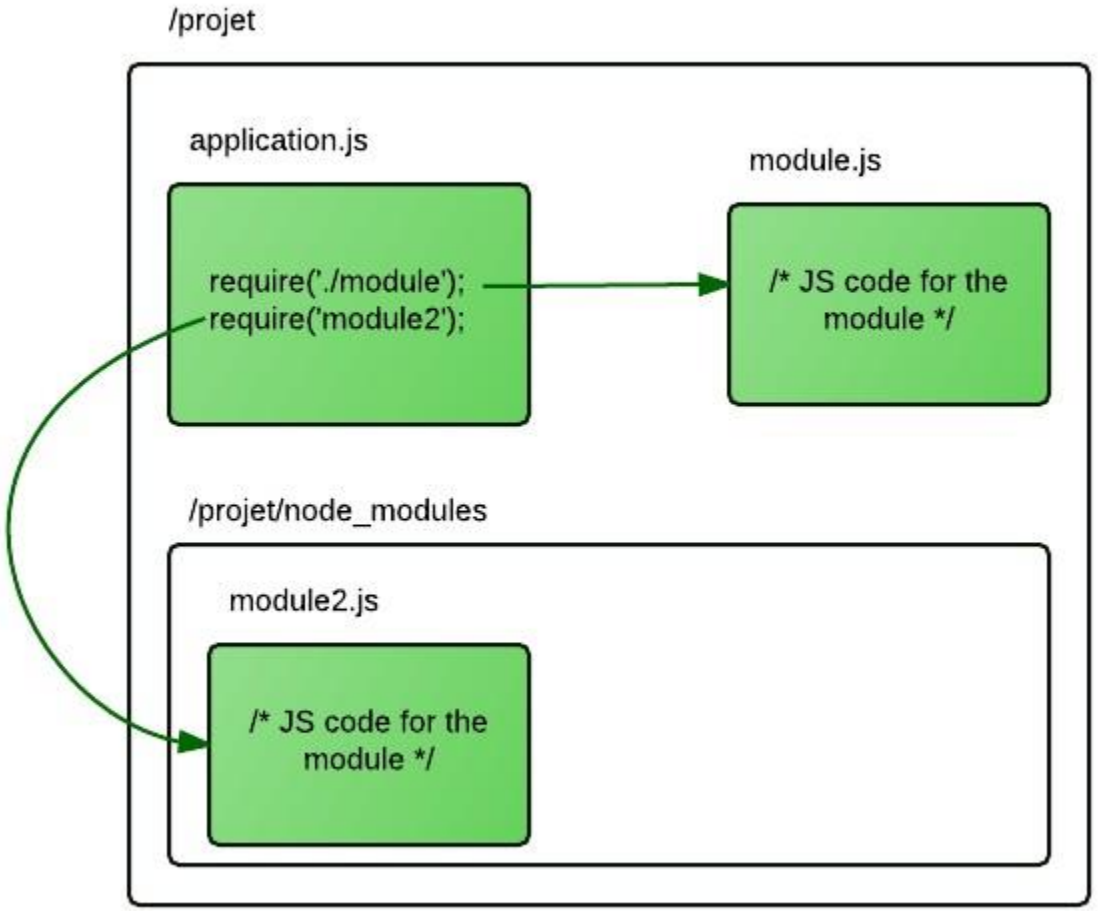
NPM & Modules

Maharishi International University - Fairfield, Iowa

Modules in NodeJS

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.

- Node implements CommonJS / **ES** Modules specs.
  - In Node.js, each file is treated as a separated module



https://openclassrooms.com

Type of Modules

- Local module (Your own Modules)
  - Simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope)*.
- Built-in Modules
  - Node.js has a set of built-in modules which you can use without any further installation.
  - buffer, fs, http, path, etc.
  - Built-in Modules Reference
- 3rd party modules on www.npmjs.com

path module

- The `path` module provides a lot of very useful functionality to access and interact with the file system.

```
import * as path from 'path';

//Return the directory part of a path:
console.log(path.dirname('File/example1.js'));

//Joins two or more parts of a path: const
name = 'joe';
console.log(path.join('users', name, 'notes.txt'));

console.log(__dirname); // returns absolute path of current file
```

fs module

- The `fs` module provides a lot of very useful functionality to access and interact with the file system.

```
import * as fs from 'fs';
import * as path from 'path';

const greet = fs.readFileSync(path.join(__dirname, 'greet.txt'), 'utf8');
console.log(greet);

fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8',
   function(err, data) { console.log(data); }
);
console.log('Done!');
```

- Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (`null` will be passed if there is no error).
- Notice: `data` here is a buffer object. We can convert it with `toString` or add the encoding - `'utf8'`

Example Read/Write Files

```
import * as fs from 'fs';
import * as path from 'path’;

// Reading from a file:
fs.readFile(path.join(__dirname, 'greet.txt'), { encoding: 'utf8' }, (err, data) =>{
    if (err) throw err;
    console.log(data);
});

// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', (err) => {
    if (err) throw err;
    console.log('Done');
});
```

Stream

- Stream is a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

- Why streams?
  - Memory efficiency: you don't need to load large amounts of data in memory before you are able to process it
  - Time efficiency: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

- The Node.js stream module provides the foundation upon which all streaming APIs are built. All streams are instances of EventEmitter

Different types of streams

- Readable: a stream you can pipe from, but not pipe into (you can receive data, but not send data to it).When you push data into a readable stream, it is buffered, until a consumer starts to read the data. (`fs.createReadStream`)

- Writable: a stream you can pipe into, but not pipe from (you can send data, but not receive from it). (`fs.createWriteStream`)

Examples of Readable and Writable streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Stream example

```
import * as fs from 'fs';
import * as path from 'path’;

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' e
vent.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk. Default is 64 kb

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'),
       { highWaterMark: 16 * 1024 });

const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.on('data', function(chunk) {
    console.log(chunk.length);
    writable.write(chunk);
});
```

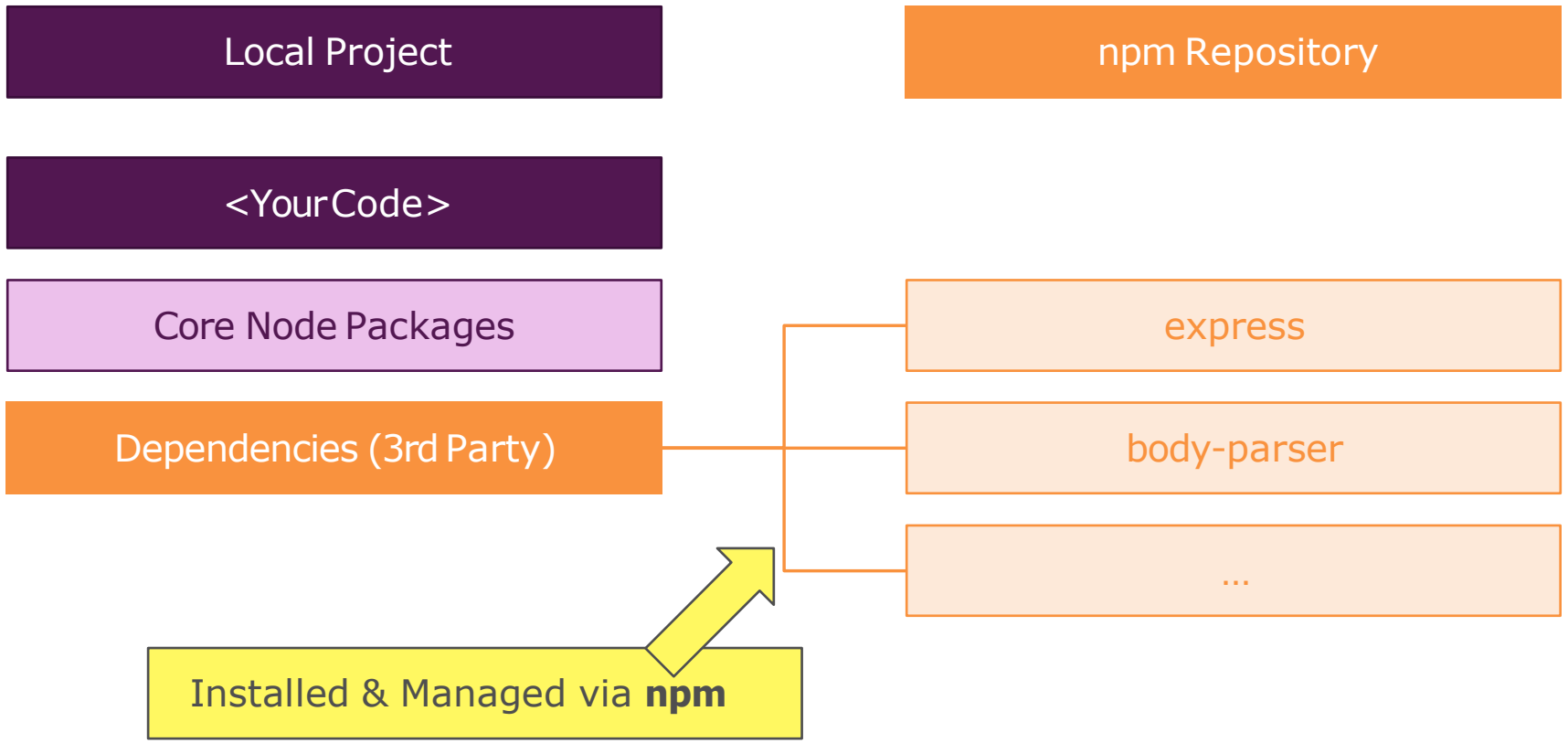
Pipes: src.pipe(dst);

- To connect two streams, Node provides a method called `pipe()` available on all readable streams. Pipes hide the complexity of listening to the stream events.

```
import * as fs from 'fs';
import * as path from 'path’;

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'));
const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.pipe(writable);

// note that pipe return the destination, this is why you can pipe it again to another
stream if the destination was readable in this case it has to be DUPLEX because you
are going to write to it first, then read it and pipe it again to another writable
stream.
```

NPM

npm & packages Intro



What is npm?

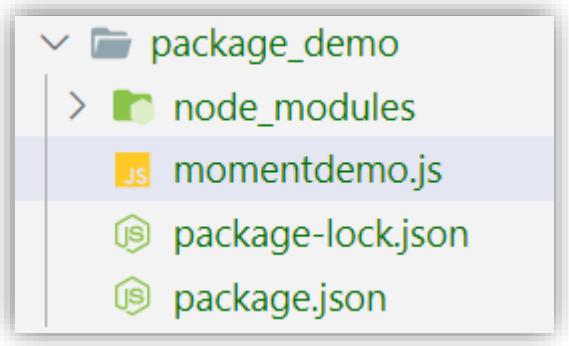
- **npm** is the standard package manager for Node.js. It also manages downloads of dependencies of your project.
- www.npmjs.com hosts thousands of free packages to download and use.
- The NPM program is installed on your computer when you install Node.js.
  - `npm -v // will print npm version`

- What is a package?
  - A package in Node.js contains all the files you need for a module.
  - Modules are JavaScript libraries you can include in your project.

  - A package contains:
    - JS files
    - `package.json` (manifest)
    - `package-lock.json` (maybe)

Create & use a new package

```
npm init // will create package.json
```



- When we install a package:
  - Notice dependencies changes in `package.json`
  - notice folder: `node_modules`
  - This structure separate our app code to the dependencies. Later when we share/deploy our application, there's no need to copy `node_modules`,
    run: `npm install` will read all dependencies and install them locally.

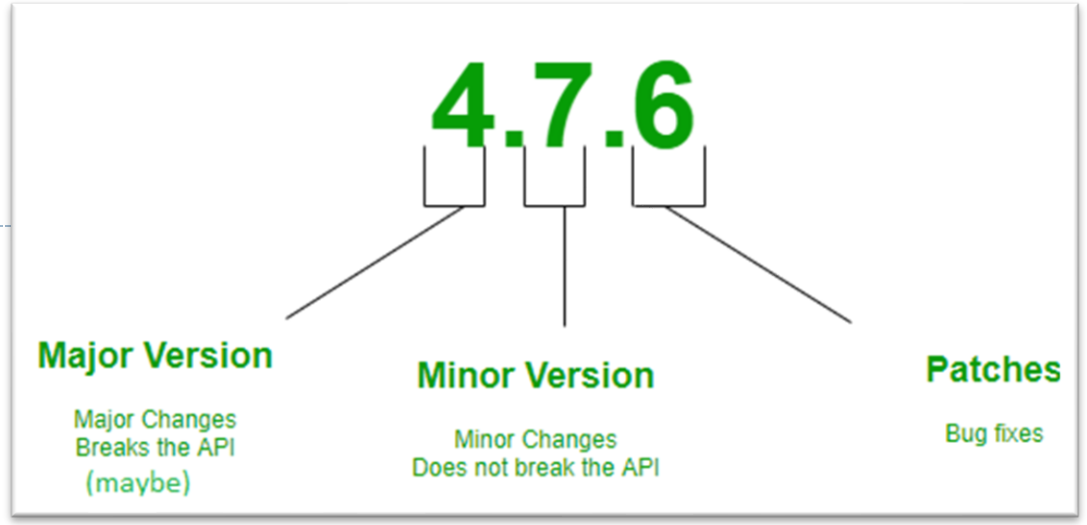
package.json Manifest

- The `package.json` file is kind of a manifest for your project.
- It can do a lot of things, completely unrelated.
- It's a central repository of configuration for installed packages.
- The only requirement is that it respects the JSON format.

- `version`: indicates the current version
- `name`: the application/package name
- `description`: a brief description of the app/package
- `main`: the entry point for the application
- `scripts`: defines a set of node scripts you can run
- `dependencies`: sets a list of npm packages installed as dependencies
- `devDependencies`: sets a list of npm packages installed as development dependencies

```
{
    "name": "package_demo",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "node momentdemo.js"
     },
    "author": "Anna Smith",
    "license": "ISC",
    "dependencies": {
        "moment": "^2.29.1"
    },
    "devDependencies": {
        "eslint": "^7.28.0"
    }
}
```

Semantic Versioning



- The Semantic Versioning concept is simple: all versions have 3 digits: `x.y.z`.
  - the first digit is the major version
  - the second digit is the minor version
  - the third digit is the patch version

- When you make a new release, you don't just up a number as you please, but you have rules:
  - you up the **major** version when you make incompatible API changes
  - you up the **minor** version when you add functionality in a backward-compatible manner
  - you up the **patch** version when you make backward-compatible bug fixes

More details about Semantic Versioning

- Why is that so important?
  - Because `npm` set some rules we can use in the `package.json` file to choose which versions it can update our packages to, when we run `npm update`.

- The rules use those symbols:
  - **^**: it's ok to automatically update to anything within this major release. If you write `^0.13.0`, when running `npm update`, it can update to `0.13.1`, `0.14.2`, and so on, but not to `1.14.0` or above.
  - **~**: if you write `~0.13.0` when running `npm update` it can update to patch releases: `0.13.1` is ok, but `0.14.0` is not.
  - **>**: you accept any version higher than the one you specify

package-lock.json

- Introduced by NPM version 5 to capture the exact dependency tree installed at any point in time.
- Describes the exact tree
- Guarantee the dependencies on all environments.
- Don’t modify this file manually.
- Always use npm CLI to change dependencies, it’ll automatically update package-lock.json

```
{
  "name": "lesson03-demo",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "moment": {
      "version": "2.24.0",
      "resolved": "https://registry.npmjs.org/moment/-
/moment-2.24.0.tgz",
      "integrity": "sha512-
bV7f+6l2QigeBBZSM/6yTNq4P2fNpSWj/0e7jQcy87A8e7o2nAfP/34/2ky5
Vw4B9S446EtIhodAzkFCcR4dQg=="
    }
  }
}
```

More About Packages

- Development Dependencies: Needed only while I'm developing the app. It's not needed for running the app.
  - `npm install mocha --save-dev`

```
// notice devDependencies entry now in package.json
```

- Global Dependencies: Available to all applications
  - `npm install -g nodemon`
  - `nodemon app.js //auto detects changes and restarts your project`

HTTP

Node as a Web Server

- Node started as a Web server and evolved into a much more generalized framework.
- Node `http` module is designed with streaming and low latency in mind.
- Node is very popular today to create and run Web servers.

Web Server Example

```
import * as http from 'http';
const server = http.createServer();

server.on('request', function(req, res) {
                        res.writeHead(200, {'Content-Type': 'text/plain'});
                        res.write('Hello World!');
                        res.end();
                    });

server.listen(3000);
```

After we run this code. The node program doesn’t stop. it keeps waiting for request

Create Web Server

```
import { createServer, IncomingMessage, Server, ServerResponse } from 'node:http';

const server: Server = createServer();

server.on('request', function(req: IncomingMessage, res: ServerResponse) {
   res.writeHead(200, {'Content-Type': 'text/plain'}); // Inform client of content type
   res.write('Hello '); // content
   res.write('World!'); // content
   res.end();
});

server.listen(3000, ()=> console.log(`listening to 3000`));

// When the server receives a request, Node creates req and res objects
// and schedules the request-handler function in the Poll queue
```

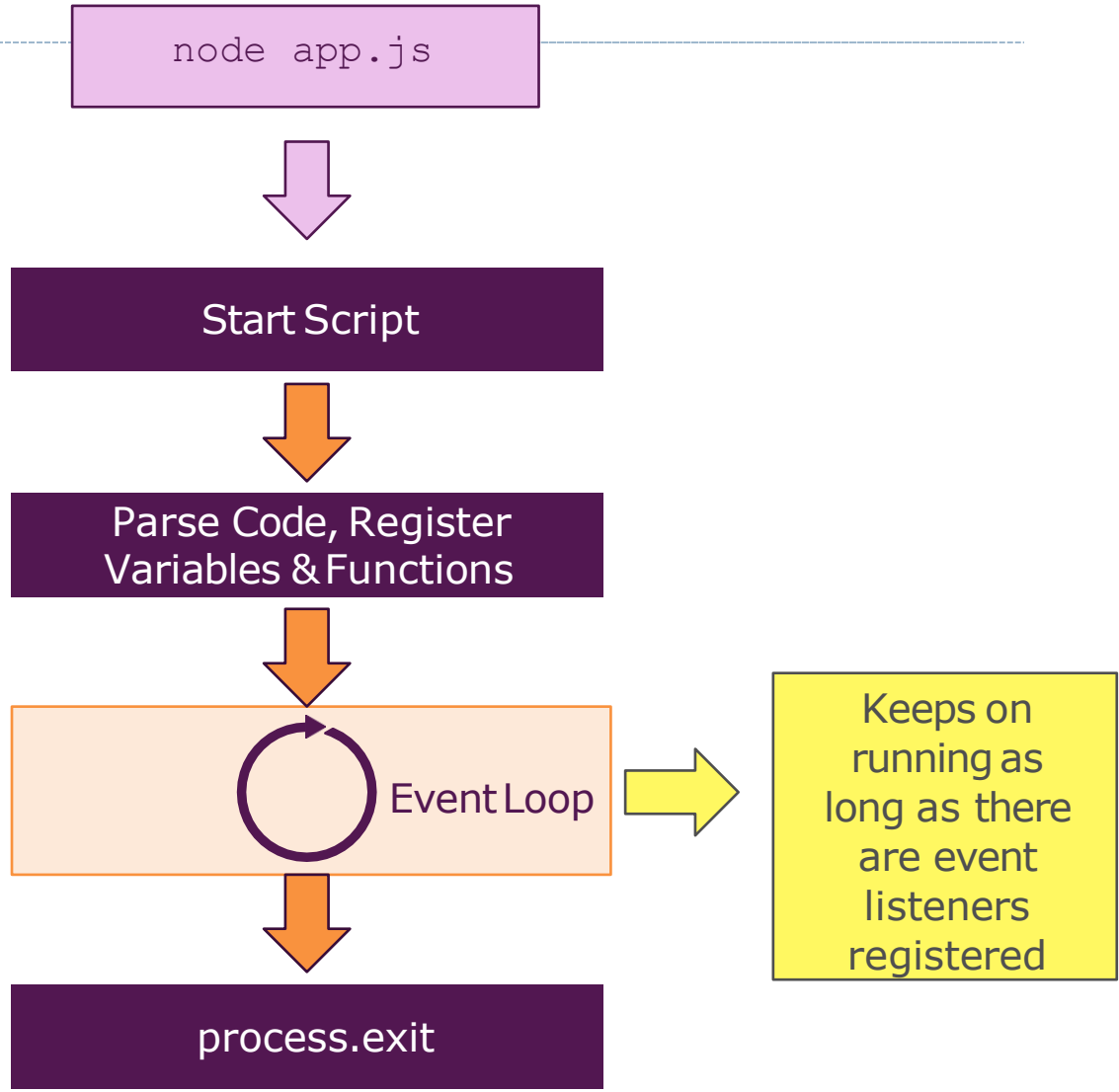
Node treats TCP Packets as Stream

Web Server Example Shortcut

Passing a callback function to `createServer()` is a shortcut for listening to "request" event.

```
import * as http from 'http';

http.createServer(function (req, res) {
     res.writeHead(200, { 'Content-Type':
                          'application/json' });
     const person =       { firstname: 'Josh',
                          lastname: 'Edward'};
     res.end(JSON.stringify(person));
}).listen(3000, '127.0.0.1');
```

The Node Application



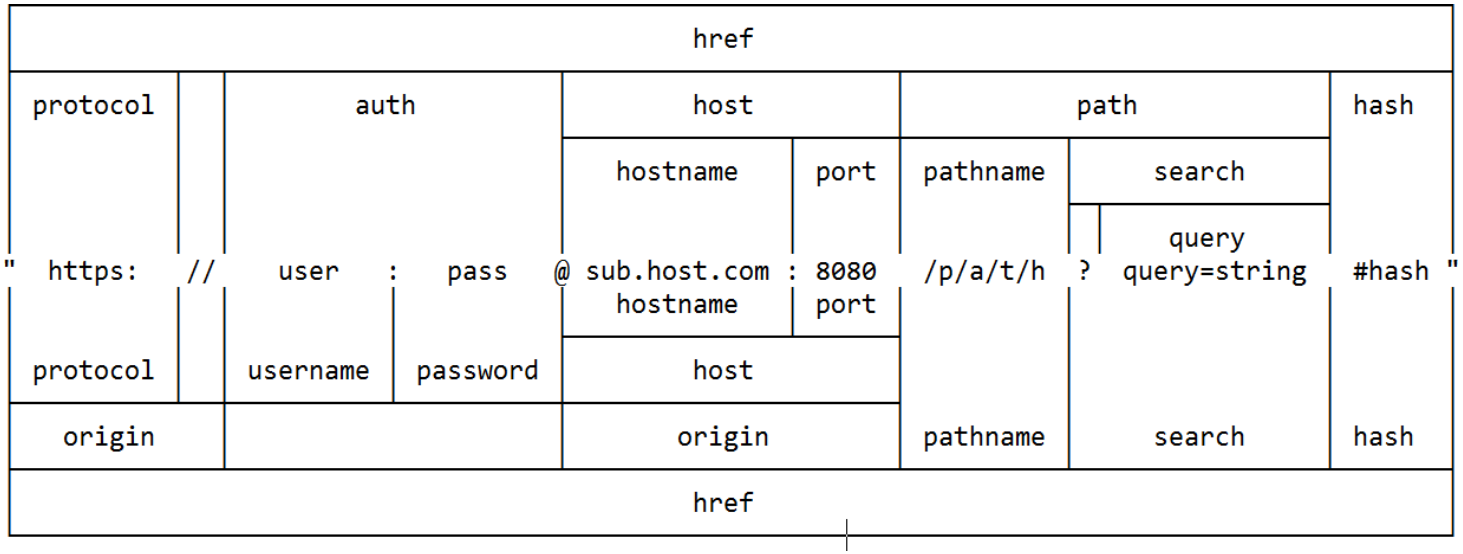
Understanding Request & Response

- A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.
- After receiving and interpreting a request message, a server responds with an HTTP response message.

```
import * as http from 'http’;

http.createServer((req, res) => {
    console.log(req.url, req.method, req.headers);
    res.setHeader('Content-Type', 'text/html');
    res.write('My First Page');
    res.write('Hello From Node.js');
    res.end();
}).listen(3000);
```

HTTP Request: Reading Get and Post Data

- Handling basic GET & POST requests is relatively simple with Node.js.
- We use the `url` module to parse and read information from the URL.
- The `url` module uses the WHATWG URL Standard (https://url.spec.whatwg.org/)



Using URL Module

- Parsing the URL string using the WHATWG API:

```
import url from 'url';
const myURL =
    url.parse('https://user:pass@sub.host.com:8080/p/a/t/h?course1=nodejs&course2=angular#hash', true);
console.log(myURL);
```

```
URL {
  href: 'https://user:pass@sub.host.com:8080/p/a/t/h?course1=nodejs&course2=angular#hash',
  origin: 'https://sub.host.com:8080',
  protocol: 'https:',
  username: 'user',
  password: 'pass',
  host: 'sub.host.com:8080',
  hostname: 'sub.host.com',
  port: '8080',
  pathname: '/p/a/t/h',
  search: '?course1=nodejs&course2=angular',
  searchParams: URLSearchParams { 'course1' => 'nodejs', 'course2' => 'angular' },
  hash: '#hash'
}
```

Parsing the Query String

```
import * as url from 'url’;

const adr= 'http://localhost:8080/default.htm?year=2017&month=february’;
const q = url.parse(adr, true);

console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2017&month=february’

let qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata.month); //returns 'february'
```