Els mòduls de Node.js


Node.js és un framework molt modularitzat, és a dir, està subdividit en nombrosos mòduls, llibreries o paquets (al llarg d'aquestes anotacions utilitzarem aquests tres termes indistintament per a referir-nos al mateix concepte). D'aquesta manera, només afegim als nostres projectes aquells mòduls que necessitem.

El propi nucli de Node.js ja incorpora algunes llibreries d'ús habitual, com per exemple la llibreria *fs* per a accedir al sistema de fitxers. A més, podem també dividir el nostre codi en diferents mòduls o fitxers interconnectats. I, finalment, podem incorporar a les nostres aplicacions altres mòduls desenvolupats per terceres parts, instal·lant-los localment. Veurem a continuació com fer cadascuna d'aquestes coses.

1. Utilitzar mòduls del nucli de Node

Com comentàvem, el propi framework Node incorpora una sèrie de mòduls pre-instal·lats que podem utilitzar en les nostres aplicacions. Podem accedir ací per a consultar els mòduls disponibles.

Per a utilitzar qualsevol mòdul (propi de Node o fet per terceres parts) en una aplicació és necessari incloure-ho en el nostre codi amb la instrucció `require`. Rep com a paràmetre el nom del mòdul a afegir, com una cadena de text.

Per exemple, crearem un arxiu anomenat `llistat.js` en el nostre projecte de "*ProjectesNode/Proves/ProvesSimples*". En ell farem un xicotet programa que utilitze el mòdul `fs` incorporat en el nucli de Node per a obtindre un llistat de tots els arxius i subcarpetas d'una carpeta determinada. El codi d'aquest arxiu pot ser més o menys així:

```
const ruta = '/Users/nacho';
const fs = require('fs');
fs.readdirSync(ruta).forEach(fichero => {console.log(fichero);});
```

Si executem aquest programa obtindrem el llistat de la carpeta indicada. Abans d'executar-ho, recorda canviar el valor de la constant `ruta` en el codi pel de una carpeta vàlida en el teu sistema.

Notar que en el codi hem declarat dues constants (`const`), en lloc de variables (`var` / `let`), ja que no necessitem manipular o modificar el codi que s'emmagatzemarà en elles una vegada carregades (no canviarem la ruta, ni el contingut del mòdul `fs` en el nostre codi). En exemples que pugueu trobar en Internet, és habitual, no obstant això, l'ús de `var` o `let` en aquests casos.

> **NOTA**: A més de la instrucció `require` per a incorporar mòduls en aplicacions Node.js, és molt habitual utilitzar la sintaxi `import` en aplicacions JavaScript en general (especialment quan s'empren frameworks client, com a Angular o React). Al llarg d'aquest curs optarem per la primera opció, però

> deixem ací indicat com s'importarien els mòduls amb `import` (encara que el seu ús comporta algunes diferències respecte a `require`):

```
import fs from 'fs';
fs.readdirSync(...);
```

> **Exercici 1:**
>
> Crea una carpeta anomenada "**SalutacioUsuari**" en el teu espai de treball, en la carpeta de "*Exercicis*". Afig un arxiu anomenat `salutació.js`. Dona una ullada en l'API de Node al mòdul `os`, i en concret al mètode `userInfo`. Utilitza-ho per a fer un programa que salude a l'usuari que ha accedit al sistema operatiu. Per exemple, si l'usuari és "may", hauria de dir "Hola may". Executa el programa en el terminal per a comprovar el seu correcte funcionament.
>
> **AJUDA:** el mètode `console.log` admet un nombre indefinit de paràmetres, i els concatena un darrere l'altre amb un espai. Així, aquestes dues línies són equivalents:
>
> ```
> console.log("Hola " + nom);
> console.log("Hola", nom);
> ```

2. Incloure els nostres propis mòduls

També podem utilitzar `require` per a incloure un arxiu el nostre dins d'un altre, de manera que podem (devem) descompondre la nostra aplicació en diferents arxius, la qual cosa la farà més fàcil de mantindre.

Per exemple, crearem un projecte anomenat "*ProvesRequire*" en la nostra carpeta de "*Proves*", amb dos arxius de moment: un anomenat `principal.js` que tindrà el codi principal de funcionament del programa, i un altre anomenat `utilitats.js` amb una sèrie de funcions o propietats auxiliars. Des de l'arxiu `principal.js`, podem incloure el d'utilitats amb la instrucció `require`, de la mateixa manera que ho vam fer abans, però indicant la ruta relativa de l'arxiu a incloure. En aquest cas quedaria així:

```
const utilitats = require('./utilitats.js');
```

És possible també suprimir l'extensió de l'arxivament en el cas d'arxius JavaScript, per la qual cosa la instrucció anterior seria equivalent a aquesta altra (Node sobreentén que es tracta d'un arxiu JavaScript):

```
const utilitats = require('./utilitats');
```

El contingut de l'arxiu `utilitats.js` ha de tindre una estructura determinada. Si, per exemple, l'arxiu té aquest contingut:

```
console.log('Entrant en utilitats.js');
```

Llavors el mer fet d'incloure'l amb `require` mostrarà per pantalla el missatge "Entrant en utilitats.js" en executar l'aplicació. És a dir, qualsevol instrucció directa que continga l'arxiu inclòs s'executa en incloure'l. El normal, no obstant això, és que aquest arxiu no continga instruccions directes, sinó una sèrie de propietats i mètodes que puguen ser accessibles des de l'arxiu que l'inclou.

Anem a això, suposem que l'arxiu `utilitats.js` té unes funcions matemàtiques per a sumar i restar dos números i retornar el resultat. Una cosa així:

```
let sumar = (num1, num2) => num1 + num2;
let restar = (num1, num2) => num1 - num2;
```

El lògic seria pensar que, en incloure aquest arxiu amb `require` des de `principal.js`, puguem accedir a les funcions `sumar` i `restar` que hem definit... però no és així.

2.1. Exportant contingut amb *module.exports*

Per a poder fer els mètodes o propietats d'un arxiu visibles des d'un altre que l'incloga, hem d'afegir-los com a elements de l'objecte `module.exports`. Així, les dues funcions anteriors s'haurien de definir d'aquesta manera:

```
module.exports.sumar = (num1, num2) => num1 + num2;
module.exports.restar = (num1, num2) => num1 - num2;
```

És habitual definir un objecte en `module.exports`, i afegir dins tot el que vulguem exportar. D'aquesta manera, tindrem d'una banda el codi del nostre mòdul, i per un altre la part que volem exportar. El mòdul quedaria així, en aquest cas:

```
let sumar = (num1, num2) => num1 + num2;
let restar = (num1, num2) => num1 - num2;

module.exports = {
    sumar: sumar,
    restar: restar
};
```

En qualsevol cas, ara sí que podem utilitzar aquestes funcions des del programa `principal.js`:

```
const utilitats = require('./utilitats');
console.log(utilitats.sumar(3, 2));
```

Notar que l'objecte `module.exports` admet tant funcions com atributs o propietats. Per exemple, podríem definir una propietat per a emmagatzemar el valor del número "pi":

```
module.exports = {
    pi: 3.1416,
    sumar: sumar,
    restar: restar
};
```

... i accedir a ella des del programa principal:

```
console.log(utilitats.pi);
```

2.2. Incloure carpetes senceres

En el cas que el nostre projecte continga diversos mòduls, és recomanable agrupar-los en carpetes, i en aquest cas és possible incloure una carpeta sencera de mòduls, seguint una nomenclatura específica. Els passos a seguir són:

- Afegir tots els mòduls (fitxers .js) que vulguem dins de la carpeta desitjada
- Crear en aqueixa carpeta un arxiu anomenat `index.js`. Aquest serà l'arxiu que s'inclourà en nom de tota la carpeta
- Dins d'aquest arxiu `index.js`, incloure (amb `require`) tots els altres mòduls de la carpeta, i exportar el que es considere.

Des del programa principal (o un altre lloc que necessite incloure la carpeta sencera), incloure el nom de la carpeta. Automàticament es localitzarà i inclourà l'arxiu `index.js`, amb tots els mòduls que aquest haja inclòs dins.

Vegem un exemple: anem a la nostra carpeta "*ProvesRequire*" creada en l'exemple anterior, i crea una carpeta anomenada "*idiomes*". Dins, crea aquests tres arxius, amb el següent contingut:

Arxiu *es.js*

```
module.exports = {
    salutacio : "Hola"
};
```

Arxiu *en.js*

```
module.exports = {
    salutacio : "Hello"
};
```

Arxiu *index.js*

```
const en = require('./en');
const es = require('./es');

module.exports = {
    es : es,
    en : en
};
```

Ara, en la carpeta arrel de "*ProvesRequire*" crea un arxiu anomenat `salutacio_idioma.js`, amb aquest contingut:

```
const idiomes = require('./idiomes');

console.log("English:", idiomes.en.salutacio);
console.log("Español:", idiomas.es.salutacio);
```

Com pots veure, des de l'arxiu principal només hem inclòs la carpeta, i amb això automàticament incloem l'arxiu `index.js` que, al seu torn, inclou als altres. Una vegada fet això, i tal com hem exportat les propietats en `index.js`, podem accedir a la salutació en cada idioma.

2.3. Incloure arxius JSON

Els arxius JSON són especialment útils, com veurem, per a definir una certa configuració bàsica (no encriptada) en les aplicacions, a més de per a enviar informació entre parts de l'aplicació (el que veurem també més endavant). Per exemple, i seguint amb l'exemple anterior, podríem traure a un arxiu JSON el text de la salutació en cada idioma. Afegim un arxiu anomenat `salutacions.json` dins de la nostra subcarpeta "idiomes":

```
{
 "es" : "Hola",
 "en" : "Hello"
}
```

Després, podem modificar el contingut dels arxius `es.js` i `en.js` perquè no posen literalment el text, sinó que l'agafen de l'arxiu JSON, incloent-lo. Ens quedarien així:

Arxiu *és.js*:

```
const textos = require('./salutacions.json');

module.exports = {
    salutació : textos.es
};
```

Arxiu *en.js*:

```
const textos = require('./salutacions.json');

module.exports = {
    salutació : textos.en
};
```

La manera d'accedir als textos des del programa principal no canvia, només l'ha fet la manera d'emmagatzemar-los, que queda centralitzada en un arxiu JSON, en lloc d'en múltiples arxius JavaScript. D'aquesta manera, davant qualsevol errata o actualització, només hem de modificar el text en l'arxiu JSON i no anar buscant arxiu per arxiu. A més, ens evita el problema de les *magic strings* (cadenes que els programadors posen a mà on toca, suposant que estan ben escrites i que no faran falta des d'una altra part de l'aplicació).

2.4. Més sobre inclusió de mòduls locals

Per a finalitzar amb aquest subapartat d'inclusió de mòduls locals de la nostra aplicació (o divisió de la nostra aplicació en diversos fitxers font, segons com vulguem veure'l), convé tindre en compte un parell de matisos addicionals:

2.4.1. Rutes relatives i *require*

Fins ara, quan hem emprat la instrucció `require` per a incloure un mòdul del nostre propi projecte, hem partit de la carpeta actual. Per exemple:

```
const utilitats = require('./utilitats');
```

Aquest codi funcionarà sempre que executem l'aplicació Node des de la seua mateixa carpeta:

```
node principal.js
```

Però si estem en una altra carpeta i executem l'aplicació des d'allí...

```
node /Users/nacho/Projectes/ProvesRequire/principal.js
```

... llavors `require` farà referència a la carpeta des d'on estem executant, i no trobarà l'arxiu "*utilitats.js*", en aquest cas. Per a evitar aquest problema, podem emprar la propietat `__dirname`, que fa referència a la carpeta del mòdul que s'està executant (`principal.js`, en aquest cas):

```
const utilitats = require(__dirname + '/utilitats');
```

2.4.2. Sobre *module.exports*

Potser alguns de vosaltres us haureu preguntat... com és que puc tindre accessibles variables o mètodes que jo no he definit en començar, com `require`, o `module.exports`?

Quan s'executa el nostre codi, Node.js ho encapsula dins d'una funció, i li passa com a paràmetres els elements externs que pot necessitar, com per exemple `require`, o `module` (en l'interior dels quals trobarem `exports`). No obstant això, en alguns exemples en Internet també podem trobar que es fa ús d'una propietat `exports`, en lloc de `module.exports`. Llavors...

- Hi ha un `exports` d'una banda i un `module.exports` per un altre? La resposta és que SI
- Existeix diferència entre tots dos? La resposta també és que SI. A priori, tots dos elements apunten al mateix objecte en memòria, és a dir, `exports` és una drecera per a no haver d'escriure `module.exports`. Però...
    - Si cometem l'error de reassignar la variable (per exemple, fent `exports = a`), llavors les referències deixen de ser iguals.
    - D'altra banda, si voleu veure el codi font de la funció `require`, veureu que el que retorna és `module.exports`, per la qual cosa, en cas de reassignar la variable `exports`, no ens serviria de res.

En resum, `exports` i `module.exports` serveixen per al mateix sempre que no les reassignem. Però durant tot aquest curs seguirem el nostre propi consell: usarem sempre `module.exports` per a evitar

problemes.

> **Exercici 2:**
>
> Per a realitzar aquest exercici, ens basarem en l'exercici "*Promeses*" de la sessió anterior, on gestionàvem les persones d'un vector mitjançant uns mètodes que inserien o esborraven dades d'aquest, i retornaven una promesa amb el resultat.
>
> Còpia aqueixa carpeta i canvia-la de nom a "**Modularitzar**". El que farem en aquest exercici és dividir el codi en dos arxius:
>
> - Un arxiu anomenat `persones.js` on definirem els dos mètodes que s'encarreguen d'afegir i esborrar persones del vector. Recorda exportar aquests mètodes amb `module.exports` per a poder-los utilitzar des de fora. També necessitaràs passar-los com a paràmetre el vector de persones, ja que aquest vector quedarà en un altre arxiu a part i no serà directament accessible.
> - Un arxiu anomenat `index.js` on inclourem el mòdul anterior. En aquest arxiu definirem el vector de persones tal com estava originalment, i el programa principal, que utilitzarà el mòdul anterior per a inserir o esborrar algunes persones de prova en el vector.
>
> Executa el programa per a verificar que les dependències amb el mòdul s'han establit correctament, i les dades s'insereixen i esborren del vector de manera satisfactòria.

3. Mòduls de tercers. El gestor *npm*

Existeixen nombrosos mòduls fets per tercers que poden ser afegits i utilitzats en les nostres aplicacions, com per exemple el mòdul *mongoose* per a accés a bases de dades MongoDB, o el mòdul *express* per a incorporar el framework Express.js al nostre projecte, i desenvolupar aplicacions web amb ell, com veurem en sessions posteriors. Aquests mòduls de tercers s'instal·len a través del gestor *npm*.

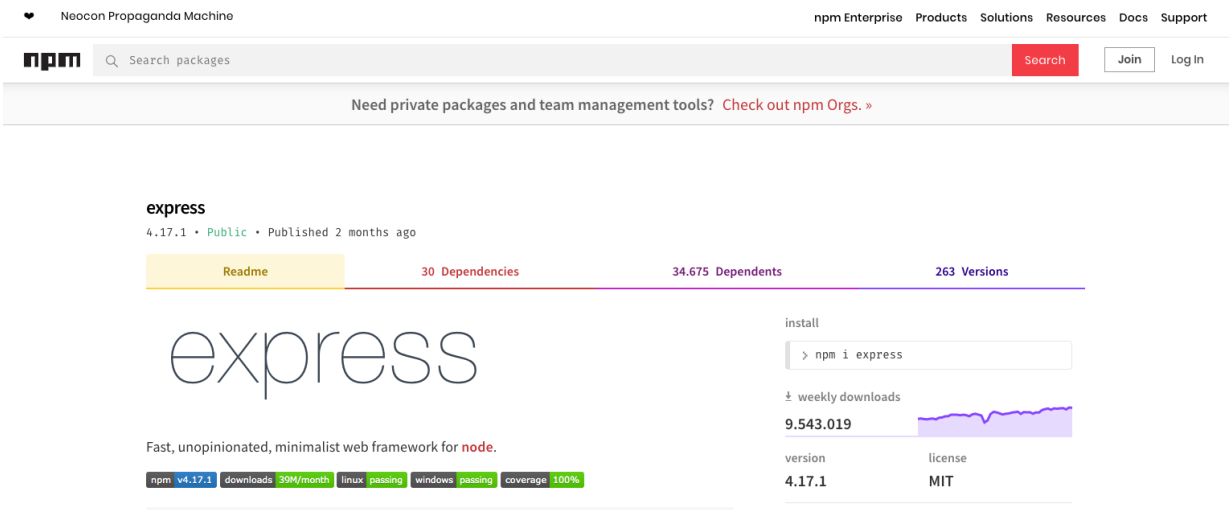
npm (*Node Package Manager*) és un gestor de paquets per a JavaScript, i s'instal·la automàticament en instal·lar Node.js. Podem comprovar que ho tenim instal·lat, i quina versió concreta tenim, mitjançant la comanda:

```
npm -v
```

encara que també ens servirà la comanda `npm --version`.

Inicialment, `npm` es va pensar com un gestor per a poder instal·lar mòduls en les aplicacions Node, però s'ha convertit en molt més que això, i a través d'ell podem també descarregar i instal·lar en les nostres aplicacions altres mòduls o llibreries que no tenen a veure amb Node, com per exemple *Bootstrap* o *jQuery*. Així que actualment és un enorme ecosistema de llibreries open-source, que ens permet centrar-nos en les necessitats específiques de la nostra aplicació, sense haver de "reinventar la roda" cada vegada que necessitem una funcionalitat que ja han fet uns altres abans. El registre de llibreries o mòduls gestionat per NPM està en la web npmjs.com.

Podem consultar informació sobre alguna llibreria en particular, consultar estadístiques de quanta gent li la descarrega, i fins i tot proporcionar nosaltres les nostres pròpies llibreries si volem. Per exemple, aquesta és la fitxa de la llibreria *express*, que emprarem més endavant:



L'opció més habitual d'ús de npm és instal·lar mòduls o paquets en un projecte concret, de manera que cada projecte tinga els seus propis mòduls. No obstant això, en algunes ocasions també ens pot interessar (i és possible) instal·lar algun mòdul de manera global al sistema. Veurem com fer aquestes dues operacions.

3.1. Instal·lar mòduls locals a un projecte

En aquest apartat veurem com instal·lar mòduls de tercers de manera local a un projecte concret. Farem proves dins d'un projecte anomenat "*ProvaNPM*" en la nostra carpeta de "*ProjectesNode/Proves*", la carpeta de les quals podem crear ja.

3.1.1. L'arxiu "package.json"

La configuració bàsica dels projectes Node s'emmagatzema en un arxiu JSON anomenat `package.json`. Aquest arxiu es pot crear directament des de línia de comandes, utilitzant una d'aquestes dues opcions (hem d'executar-la en la carpeta del nostre projecte Node):

- `npm init --yes`, que crearà un arxiu amb uns valors per defecte, com aquest que es mostra a continuació:

```
{
    "name": "ProvaNPM",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "tire \"Error: no test specified\" && exit 1"
    },
    "keywords": [],
    "author": "",
    "license": "ISC"
}
```

- `npm init`, que iniciarà un assistent en el terminal perquè donem valor a cada atribut de la configuració. El més típic és emplenar el nom del projecte (presa com a valor per defecte el nom de la carpeta on està), la versió, l'autor i poc més. Moltes opcions tenen valors per defecte posats entre parèntesi, per la qual cosa si premem *Intro* s'assignarà aquest valor sense més.

Al final de tot el procés, tindrem l'arxiu en la carpeta del nostre projecte. En ell afegirem després (de manera manual o automàtica) els mòduls que necessitem, i les versions d'aquests, com explicarem a continuació.

> **NOTA**: en generar l'arxiu `package.json`, podem observar que el nom de programa principal (*entry point*) que s'assigna per defecte a l'aplicació Node és *index.js*. És habitual que el fitxer principal d'una aplicació Node es diga així, o també *app.js*, com veurem en posteriors exemples, encara que no és obligatori cridar-los així.

3.1.2. Afegir mòduls al projecte i utilitzar-los

Per a instal·lar un mòdul extern en un projecte determinat, hem d'obrir un terminal i situar-nos en la carpeta del projecte. Després, escrivim la següent comanda:

```
npm install nom_modul
```

on *nom_modul* serà el nom del mòdul que vulguem instal·lar. Podem instal·lar també una versió específica del mòdul afegint-lo com a sufix amb una arrova al nom del mòdul. Per exemple:

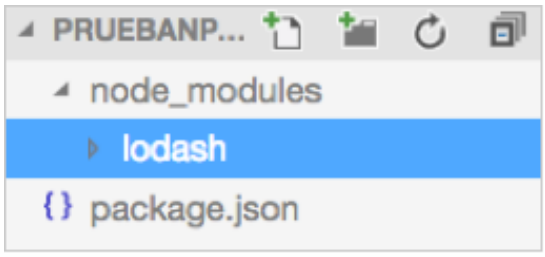
```
npm install nom_modul@1.1.0
```

Provarem amb un mòdul senzill i molt utilitzat (té milions de descàrregues setmanalment), ja que conté una sèrie d'utilitats per a facilitar-nos el desenvolupament dels nostres projectes. Es tracta del mòdul `lodash`, que podeu consultar en la web citada anteriorment (ací). Per a instal·lar-ho, escrivim el següent:

```
npm install lodash
```

Algunes puntualitzacions abans de seguir:

- Després d'executar la comanda anterior, s'haurà afegit el nou mòdul en una subcarpeta anomenada `node_modules` dins del nostre projecte i a l'arxiu `package.json`.




```
{
    "name": "provanpm",
    ...
    "dependencies": {
        "lodash": "^4.17.15"
    }
}
```

- En instal·lar qualsevol nou mòdul, es generarà (o modificarà) un arxiu addicional anomenat `package-lock.json`. Aquest arxiu és una còpia de seguretat de com ha quedat l'arbre de carpetes en "*node_modules*" amb la nova instal·lació, de manera que puguem tornar arrere i deixar els mòduls com estaven en qualsevol pas previ. És utilitzat en repositoris *git* per a aquestes restauracions, precisament. Nosaltres no li farem molt cas de moment.

Per a poder utilitzar el nou mòdul, procedirem de la mateixa forma que per a utilitzar mòduls predefinits de Node: emprarem la instrucció `require` amb el nom original del mòdul. Per exemple, editarem un arxiu `index.js` en la carpeta "*ProvaNPM*" que venim editant en aquests últims passos, i afegim aquest codi que carrega el mòdul "lodash" i l'utilitza per a eliminar un element d'un vector:

```
const lodash = require('lodash');
console.log(lodash.difference([1, 2, 3], [1]));
```

> **NOTA:** si busqueu documentació o exemples d'ús d'aquesta llibreria en Internet, és habitual que el nom de variable o constant on es carrega (en la línia `require`) siga un simple símbol de subratllat (això és el que significa *low dash* en anglés), amb el que l'exemple anterior quedaria així:

```
const _ = require('lodash');
console.log(_.difference([1, 2, 3], [1]));
```

Si executem aquest exemple des del terminal, obtindrem el següent:

```
node index.js
[ 2, 3 ]
```

Exercici 3:

Crea una carpeta anomenada "**Lodash**" en el teu espai de treball, en la carpeta de "*Exercicis*". Dins, crea un arxiu `package.json` utilitzant la comanda `npm init` vist abans. Deixa els valors per defecte que et planteja l'assistent, i posa el teu nom com a autor.

Després, instal·la el paquet `lodash` com s'ha explicat en un exemple anterior, i consulta la seua documentació ací, per a fer un programa en un arxiu `index.js` que, donat un vector de noms de persones, els mostre per pantalla separats per comes. Hauràs de definir a mà l'array de noms dins del codi. Per exemple, per a l'array `["Nacho", "Ana", "Mario", "Laura"]`, l'eixida del programa haurà de ser:

```
Nacho,Ana,Mario,Laura
```

NOTA: revisa el mètode `join` dins de la documentació de "lodash", pot ser-te molt útil per a aquest exercici.

3.1.3. Desinstal·lar un mòdul

Per a desinstal·lar un mòdul (i eliminar-lo de l'arxiu `package.json`, si existeix), escrivim la comanda següent:

```
npm uninstall nom_modul
```

3.1.4. Ordre d'inclusió dels mòduls

Hem vist com incloure en una aplicació Node tres tipus de mòduls:

- Mòduls pertanyents al nucli de Node
- Mòduls del nostre propi projecte, si està dividit en diversos fitxers font
- Mòduls fets per tercers, descarregats a través de npm

Encara que no hi ha una norma obligatòria a seguir sobre aquest tema, sí que és habitual que, quan la nostra aplicació necessita incloure mòduls de diversos tipus (predefinits de Node, de tercers i arxius propis), es faça amb una estructura determinada.

Bàsicament, el que es fa és incloure primer els mòduls de Node i els de tercers, i després (separats per un espai del bloc anterior), els arxius propis del nostre projecte. Per exemple:

```
const fs = require('fs');
const _ = require('lodash');

const utilitats = require('./utilitats');
```

3.1.5. Algunes consideracions sobre mòduls de tercers

Hem vist els passos elementals per a poder instal·lar, utilitzar, i desinstal·lar (si és necessari) mòduls de tercers localment en les nostres aplicacions. Però hi ha alguns aspectes referents a aquests mòduls, i la forma en què s'instal·len i distribueixen, que has de tindre en compte.

Gestió de versions

Des que comencem a desenvolupar una aplicació fins que la finalitzem, o en manteniments posteriors, és possible que els mòduls que la componen s'hagen actualitzat. Algunes d'aqueixes noves versions poden no ser compatibles amb el que en el seu moment vam fer, o al contrari, hem actualitzat l'aplicació i ja no ens serveixen versions massa antigues d'uns certs mòduls.

Per a poder determinar quines versions o rangs de versions són compatibles amb el nostre projecte, podem utilitzar la mateixa secció de "*dependencies*" de l'arxiu `package.json`, amb una nomenclatura determinada. Vegem alguns exemples utilitzant el paquet "*lodash*" del cas anterior:

- `"lodash": "1.0.0"` indicaria que l'aplicació només és compatible amb la versió 1.0.0 de la llibreria
- `"lodash": "1.0.x"` indica que ens serveix qualsevol versió 1.0
- `"lodash": "*"` indica que volem tindre sempre l'última versió disponible del paquet. Si deixem una cadena buida "", es té el mateix efecte. No és una opció recomanable en alguns casos, al no poder controlar el que conté aqueixa versió.
- `"lodash": "> = 1.0.2"` indica que ens serveix qualsevol versió a partir de la 1.0.2
- `"lodash": "< 1.0.9"` indica que només són compatibles les versions de la llibreria fins a la 1.0.9 (sense comptar aquesta última).
- `"lodash": "^1.1.2"` indica qualsevol versió des de la 1.1.2 (inclusivament) fins al següent salt major de versió (2.0.0, en aquest cas, sense incloure aquest últim).
- `"lodash": "~1.3.0"` indica qualsevol versió entre la 1.3.0 (inclusivament) i la següent versió menor (1.4.0, exclusivament).

Existeixen altres modificadors també, però amb aquests podem fer-nos una idea del que podem controlar. Una vegada hàgem especificat els rangs de versions compatibles de cada mòdul, amb la següent comanda

actualitzem els paquets que es vegen afectats per aquestes restriccions, deixant per a cadascun una versió dins del rang compatible indicat:

```
npm update --save
```

Afegir mòduls a mà en "package.json"

També podríem afegir a mà en l'arxiu "package.json" mòduls que necessitem instal·lar. Per exemple, així afegiríem a l'exemple anterior l'última versió del mòdul "express":

```
{
    ...
    "dependencies": {
        "lodash": "^4.17.4",
        "express": "*"
    }
}
```

Per a fer efectiva la instal·lació dels mòduls d'aquest arxiu, una vegada afegits, hem d'executar aquesta comanda en el terminal:

```
npm install
```

Automàticament s'afegiran els mòduls que falten en la carpeta "node_modules" del projecte.

Compartir el nostre projecte

Si decidim pujar el nostre projecte a algun repositori en Internet com Github o similars, o deixar que algú li ho descarregue per a modificar-ho després, no és bona idea pujar la carpeta "node_modules", ja que conté codi font fet per terceres persones, provat en entorns reals i fiable, que no hauria de poder-se modificar a la lleugera. A més, la forma en què s'estructura la carpeta "node_modules" depén de la versió de npm que cadascun tinguem instal·lada, i és possible que ocupe massa. De fet, els propis mòduls que descarreguem poden tindre dependències amb altres mòduls, que al seu torn es descarregaran en una subcarpeta interna.

Per tant, el recomanable és no compartir aquesta carpeta (no pujar-la al repositori, o no deixar-la a terceres persones), i no és cap problema fer això, ja que gràcies a l'arxiu `package.json` sempre podem (devem) executar la comanda:

```
npm install
```

i descarregar totes les dependències que en ell estan reflectides. Dit d'una altra forma, l'arxiu `package.json` conté un resum de tan extern com el nostre projecte necessita, i que no és recomanable facilitar amb aquest.

3.2. Instal·lar mòduls globals al sistema

Per a una certa mena de mòduls, especialment aquells que s'executen des de terminal com Grunt (un gestor i automatizador de tasques JavaScript) o JSHint (un comprovador de sintaxi JavaScript), pot ser interessant instal·lar-los de manera global, per a poder-los usar dins de qualsevol projecte.

La manera de fer això és similar a la instal·lació d'un mòdul en un projecte concret, afegint algun paràmetre addicional, i amb la diferència que, en aquest cas, no és necessari un arxiu "package.json" per a gestionar els mòduls i dependències, ja que no són mòduls d'un projecte, sinó del sistema. La sintaxi general de la comanda és:

```
npm install -g nom_modul
```

on el flag `-g` fa referència al fet que es vol fer una instal·lació global.

És important, a més, tindre present que qualsevol mòdul instal·lat de manera global en el sistema no podrà importar-se amb `require` en una aplicació concreta (per a fer-ho hauríem d'instal·lar-lo també de manera local a aquesta aplicació).

3.2.1. Exemple: nodemon

Vegem com funciona la instal·lació de mòduls a nivell global amb un realment útil: el mòdul `nodemon`. Aquest mòdul funciona a través del terminal, i ens serveix per a monitorar l'execució d'una aplicació Node, de manera que, davant qualsevol canvi en aquesta, automàticament la reinicia i torna a executar-la per nosaltres, evitant-nos haver d'escriure la comanda `node` en el terminal de nou. Podeu consultar informació sobre nodemon ací.

Per a instal·lar `nodemon` de manera global escrivim la següent comanda (amb permisos d'administrador):

```
npm install -g nodemon
```

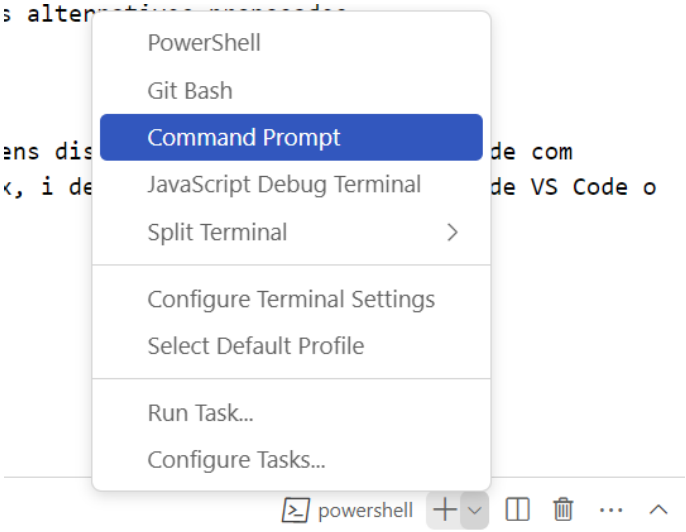
En instal·lar-ho de manera global, s'afegirà la comanda `nodemon` en la mateixa carpeta on resideixen les comandes `node` o `npm`. Per a utilitzar-ho, n'hi ha prou amb col·locar-nos en la carpeta del projecte que vulguem provar, i emprar aquesta comanda en lloc de `node` per a llançar l'aplicació:

```
nodemon index.js
```

Automàticament apareixeran diversos missatges d'informació en pantalla i el resultat d'executar el nostre programa. Davant cada canvi que fem, es reiniciarà aquest procés tornant a executar-se el programa.

Per a finalitzar l'execució de `nodemon` (i, per tant, de l'aplicació que estem monitorant), n'hi ha prou amb prémer Control+C en el terminal.

> **NOTA**: si utilitzem el terminal *powershell* de Visual Studio Code podem tindre problemes per a executar *Nodemon*. En aquest cas hem de configurar el terminal perquè siga de tipus *Command Prompt*



3.2.2. Desinstal·lar mòduls globals

De la mateixa manera, per a desinstal·lar un mòdul que s'ha instal·lat de manera global, utilitzarem la comanda:

```
npm uninstall -g nom_module
```

> **Exercici 4:**
>
> Crea una carpeta anomenada "**Moment**" en el teu espai de treball, en la carpeta "*Exercicis*". Dins, crea un arxiu `package.json` amb la corresponent comanda `npm init` vist abans. Deixa els valors per defecte que et planteja l'assistent, i posa el teu nom com a autor.
>
> Instal·la el paquet `moment`, una llibreria per a gestió de dates i temps la documentació dels quals es pot consultar ací. Defineix un programa principal en un arxiu `index.js` que incloga aquesta llibreria:

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

Una vegada inclosa, fes el següent:

- Guarda en una variable la data i hora actuals. Això pots fer-ho amb:

```
let ara = moment();
```

- Defineix una data anterior a l'actual. Pots especificar el format de la data en una cadena de text, seguit del patró d'aquesta data. Per exemple:

```
let abans = moment("07/10/2015", "DD/MM/YYYY");
```

- Defineix també una data posterior a l'actual. Pots utilitzar la mateixa nomenclatura que per a la data anterior, però amb una posterior.
- Imprimeix per consola quants anys han passat des de la data vella a l'actual. Per a calcular aquesta dada, et pot ser d'utilitat el mètode `duration`:

```
console.log(moment.duration(dataPosterior.diff(dataAnterior)).years());
```

- Saca per consola, d'una forma similar, quants anys i mesos falten per a arribar a la data futura des de l'actual.
- Mostra ara per consola si la data vella és, efectivament, anterior a l'actual. Per a això pots utilitzar el mètode `isBefore` (o `isAfter`, depenent de com les compares):

```
if (dataAnterior.isBefore(dataPosterior))...
if (dataPosterior.isAfter(dataAnterior))...
```

- Finalment, crea una data que siga exactament dins d'un mes. Per a això, usa el mètode `add`, afegint un mes a la data actual. Saca aquesta data per pantalla, formatada com *DD/MM/YYYY*. Utilitza el mètode `format` per a això.

Si no ho has fet encara, instal·la el mòdul `nodemon` de manera global al sistema, com s'ha explicat en aquesta sessió. Executa aquesta aplicació amb aquest mòdul, i comprova que totes les dades que es mostren per consola són els esperats. Després, prova de canviar alguna data (la passada i/o la futura), i comprova com s'executa de nou automàticament i mostra els nous resultats.