Desarrollo de paquetes

Tiempo estimado: 30min

En la lección anterior, aprendimos cómo usar paquetes de NPM. Ahora, ha llegado el momento de describir cómo desarrollar nuestros propios paquetes. Pieza clave del desarrollo de software con Node.

Comenzamos con un breve recordatorio sobre la reutilización de código en Node. A continuación, describimos la estructura de directorios que debe presentar un paquete. Después, mostramos el archivo package.json, indispensable en todo paquete. Luego, presentamos las dependencias y el módulo principal de un paquete. Finalmente, el archivo .npmignore y el comando npm link.

Al finalizar la lección, el estudiante sabrá:

- Cómo desarrollar un paquete personalizado.
- Para qué sirve el archivo package.json.
- Cómo indicar las dependencias del paquete.
- Cómo instalar las dependencias localmente.
- Cómo fijar el módulo principal o punto de entrada del paquete.
- Cómo usar el comando npm link para crear un enlace al paquete en el directorio predeterminado global.

Introducción

En Node, la reutilización de software o reutilización de código se consigue mediante paquetes y módulos. Sabemos que un paquete (*package*) es un directorio de código reutilizable. Mientras que un módulo (*module*) es un archivo de código JavaScript reutilizable. Ambos tipos de componentes disponen de una API, cuya funcionalidad está clara y bien definida. Ambos se reutilizan mediante la función require() y la sentencia import de JavaScript.

Estructura de un paquete

De la lección anterior, sabemos que un paquete no es más que un directorio de código JavaScript y otros archivos. Node no requiere ninguna estructura de directorios específica aunque, como casi todo en programación, hay convenios. Para evitar que cada paquete tenga una estructura particular, según la manera de pensar o concebir las cosas de cada programador, se utilizan los convenios. Con los convenios, se busca que todos parezcamos uno.

En el caso de Node, por lo general, el código del paquete se estructura como sigue:

- Archivo package.json. Este archivo, obligatorio en todo paquete Node, contiene los metadatos del paquete.
- Archivo index.js. Punto de entrada del paquete. El archivo que ejecuta el motor de Node cuando carga el paquete por primera vez. Define su API. Lo que exporta este módulo, es lo que exporta el paquete.

  Se puede indicar otro mediante la propiedad main del archivo package.json.
- Archivo .npmrc del proyecto. Contiene el archivo de configuración npm específico del paquete.

  No es necesario a menos que tengamos una configuración particular de npm para el paquete.
- Archivo .npmignore. Contiene archivos que no se consideran parte real del paquete NPM.
- Archivo README.md. Archivo, en formato Markdown, que contiene información para el usuario del paquete.
- Directorio bin/. Contiene los archivos que deben desplegarse en el directorio bin del directorio

predeterminado.

- Directorio conf/. Directorio con las configuraciones del paquete para los distintos entornos.

  Por ejemplo, el archivo production.js contendrá la específica del entorno de producción, mientras que development.js, la de desarrollo. Generalmente, cuando se genere el archivo empaquetado, se determina qué archivo de configuración usar. Por buenas prácticas, no se recomienda adjuntar la configuración de producción y desarrollo en el paquete publicado. Sólo la de producción.
- Directorio lib/. Directorio con el código JavaScript específico del paquete.
- Directorio scripts/. Contiene los *scripts* usados en la propiedad scripts del archivo package.json.
- Directorio test/unit/. Contiene las pruebas de unidad del paquete.

Además de estas entradas, también suele haber otras como, por ejemplo:

- Archivo .editorconfig. Es un tipo de archivo común que contiene parámetros de configuración para los editores o IDEs de desarrollo.
- Archivo .eslintrc con las reglas ESLint para los archivos JavaScript del proyecto.
- Archivo .eslintignore con los archivos y directorios que debe ignorar ESLint.
- Archivo .gitignore que contiene los archivos y directorios a ignorar de cara al control de versiones de Git.
- Archivo .travis.yml para la configuración de pruebas en Travis CI.
- Archivo Justo.js con el catálogo de tareas automatizadas mediante Justo.js.

  Este archivo se utiliza cuando se usa Justo como herramienta de automatización. Puede utilizar otras herramientas de este tipo como Grunt o Gulp.
- Directorio dist/ en el que depositar la versión del paquete a publicar en el repositorio o registro NPM.

Generador justo-generator-node

Para facilitar el arranque del proyecto, podemos utilizar el generador de Node de Justo.js. Justo es una herramienta de automatización que ayuda a automatizar tareas manuales para mejorar principalmente el rendimiento y la productividad. Groso modo, un generador (*generator*) no es más que un componente que crea o produce algo, en nuestro caso, la estructura inicial de un proyecto Node.

Para utilizarlo, es necesario tener instalado la aplicación CLI de Justo.js y el generador de Node. Ambas cosas, se instalan fácilmente mediante NPM y se recomienda hacerlo globalmente:

```
npm install -g justo-cli justo-generator-node
```

Una vez instalados Justo y el generador, no tenemos más que pedirle a Justo que ejecute el generador de Node, lo que desplegará una pequeña batería de preguntas y, con la información recopilada del usuario, generará la estructura inicial del proyecto:

```
justo -g node
```

Una vez ejecutado el generador, se puede consultar la estructura de directorios del proyecto en el directorio actual.

Archivo package.json

El archivo package.json es un archivo de texto en formato JSON que contiene metadatos del paquete como, por ejemplo, su nombre, su versión, su descripción, su autor, la lista de colaboradores, el archivo que actúa como punto de entrada del paquete, las dependencias, etc. Todo paquete debe tener uno y debe encontrarse en el directorio del paquete.

Este archivo contiene varias propiedades. Actualmente, sólo dos son obligatorias: name y version. Todas las demás son opcionales. La lista completa de propiedades se encuentra en el sitio web oficial de NPM, concretamente, en docs.npmjs.com/files/package.json. He aquí una lista de las propiedades más utilizadas:

- name (string). Nombre del paquete. Debe ser único en el repositorio en el que se publique.
- version (string). Versión del paquete. Debe ser distinta en cada publicación del paquete.
- homepage (string). URL del sitio web oficial del paquete.
- description (string). Breve descripción del paquete.

  Se muestra en las búsquedas de paquetes contra el repositorio en el que se encuentra publicado.
- keywords (string[]). *Array* de palabras claves que describen el paquete.
- author (string u object). Información sobre el autor del paquete.
- contributors (string[] u object[]). *Array* de cadenas de texto o de objetos que enumera los colaboradores del proyecto.
- main (string). Archivo principal del proyecto.

  Módulo del paquete que debe cargar node cuando se importe el paquete por primera vez. Lo que exporta este módulo forma la API reutilizable del paquete.

  Por convenio y buenas prácticas, se usa el módulo index.js.
- files (string[]). Archivos y directorios que forman parte del paquete final.
- dependencies (object). Dependencias de producción del paquete.
- devDependencies (object). Dependencias de desarrollo del paquete.
- engines (object). Dependencias del motor de ejecución.
- os (string[]). Dependencias de sistema operativo.
- cpu (string). Dependencias de arquitectura.
- bin (object). Archivos que deben crearse en el directorio bin del directorio donde se instala.
- repository (object). Repositorio donde se puede encontrar el código fuente del proyecto:
  - type (string). Tipo de repositorio. Generalmente, git.
  - url (string). URL donde se encuentra el repositorio.

  Generalmente hace referencia al repositorio Git del paquete.
- bugs (object). Información para comunicar *bugs* del paquete al equipo de desarrollo. El objeto puede tener dos propiedades:
  - url (string). URL de la página web a través de la cual hacerlo.
  - email (string). Dirección de correo electrónico a través de la cual hacerlo.
- preferGlobal (boolean). Indica si el autor recomienda que el paquete se instale globalmente.

  Es sólo una sugerencia. No una obligación.
- private (boolean). Indica si se trata de un paquete privado, no publicable.

  El comando npm rechaza publicar paquetes privados.
- scripts (object). Líneas de comando que pueden invocarse mediante el comando npm run.

Identificador del paquete

Todo paquete se identifica de manera única mediante la concatenación de dos propiedades: name y version. La propiedad name indica su nombre. Mientras que version, su versión. Esta versión debe seguir la sintaxis de versiones semánticas definida en semver.org:

```
major.minor.patch
```

Como, por ejemplo, 2.5.1.

major indica una versión que contiene cambios incompatibles con las anteriores. Cada vez que haga un cambio de la API que pueda ser incompatible con las anteriores, debe incrementar este valor. minor se

incrementa cuando se añade *nueva funcionalidad* al paquete, pero *no* presenta incompatibilidades con las anteriores. Finalmente, patch se debe incrementar cuando no se añada ninguna nueva funcionalidad, resuelve *bugs* o defectos.

Autores y colaboradores

El autor (*author*) contiene información sobre la persona u organización que comenzó o que ha producido el paquete. Un colaborador (*contributor*) es una persona que contribuye al desarrollo o el mantenimiento del paquete. Esta información se puede proporcionar, y se recomienda hacerlo, mediante las propiedades author y contributors, respectivamente.

Cada persona u organización se puede registrar mediante una cadena de texto con alguna de las siguientes sintaxis:

```
Nombre
Nombre <correo>
Nombre <correo> (web)
Nombre (web)
```

Otra forma es mediante un objeto con las siguientes propiedades:

- name (string). Nombre de la persona u organización.
- email (string). Correo electrónico de la persona u organización.
- url (string). URL del sitio web de la persona u organización.

He aquí unos ejemplos:

```
//mediante texto
"author": "Raúl G. <raulg@nodemy.com> (http://nodemy.com)"

//mediante objeto
"author": {
  "name": "Raúl G.",
  "email": "raulg@nodemy.com",
  "url": "http://nodemy.com"
}
```

Archivos a copiar en las instalaciones

Los archivos y directorios que debe copiar npm install cuando instale el paquete se indican en la propiedad files. Siempre, aunque no se indique, copiará los siguientes archivos:

- package.json.
- README, en cualquiera de sus variantes.
- CHANGELOG, en cualquiera de sus variantes.
- LICENSE o LICENCE.

Cualquier otro archivo o directorio que deba copiarse durante la instalación, hay que indicarlo mediante la propiedad files. Por ejemplo, para pedir que copie la carpeta lib y template, tendríamos que incluir, en package.json, lo siguiente:

```
"files": [
  "lib/",
  "template/"
]
```

Dependencias de los motores de ejecución

En algunas ocasiones, los paquetes requieren una determinada versión de los comandos node y/o npm. Si éste es el caso, se utiliza la propiedad engines. Consiste en un objeto donde cada par clave-valor representa un comando. La clave indica el motor: node o npm. Mientras que el valor, la versión del motor que necesita el paquete para una correcta ejecución.

Ejemplo:

```
"engines": {
  "node": ">=6.9.0",
```



```
  "npm": ">=3.10.0"
}
```

Dependencias de plataforma

Aunque Node es multiplataforma, algunos paquetes sólo pueden ejecutarse en determinadas plataformas, por ejemplo, por dependencias nativas.

Por un lado, se puede indicar el sistema operativo en el que el paquete funciona correctamente, mediante la propiedad os. Un *array* donde cada elemento indica un sistema operativo. Ejemplo:

```
"os": [
  "linux",
  "win32"
]
```

El valor de process.platform debe encontrarse en esta lista.

Si el nombre se precede de un signo de exclamación (!), se considera cualquier plataforma salvo la indicada. Veamos un ejemplo ilustrativo:

```
"os": ["!win32"]
```

Por otro lado, tenemos la propiedad cpu, con la que fijar las arquitecturas en las que se puede ejecutar el paquete. El valor de process.arch debe encontrarse en la lista indicada por cpu. Ejemplo ilustrativo:

```
"cpu": [
  "x64",
  "ia32"
]
```

Instalación local de paquetes

Como sabemos, la instalación de un paquete es la operación mediante la cual instalamos un paquete NPM, en nuestra máquina, para su reutilización. Hemos presentado la instalación global como aquella que instala los paquetes en un determinado directorio común, conocido como directorio predeterminado. Los paquetes en ese directorio se pueden reutilizar en cualquier proyecto. Se instalan una vez, se reutilizan varias veces.

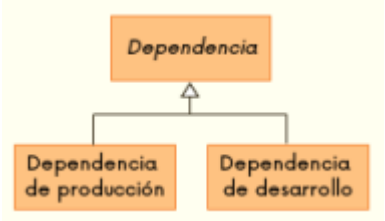
En cambio, la instalación de paquetes local es aquella que instala el paquete en un directorio específico del proyecto. Un paquete instalado localmente tiene como objeto instalarlo a nivel de aplicación particular. Esto hace que sólo ese proyecto lo pueda reutilizar. La ventaja principal es que el mismo paquete se puede instalar en varios paquetes y hacerlo con versiones distintas. Así, un proyecto P puede importar la versión V de un paquete y otro proyecto Q, la versión W. Pudiendo estas versiones ser incompatibles entre sí.

La instalación local es mucho más utilizada que la global.

La instalación local también se realiza con el comando npm install, pero *sin* la opción -g. Cuando usemos la opción -g, realizaremos instalaciones globales, en el directorio lib/node_modules del directorio predeterminado. Mientras que cuando omitamos la opción, lo haremos en el directorio node_modules del directorio del paquete.

Dependencias

Una dependencia (*dependency*) representa una relación de uso de un determinado paquete en el paquete. Los dos tipos de dependencia más habituales son la de producción y la de desarrollo.



Veamos cada una detenidamente.

Dependencias de producción

Una dependencia de producción (*production dependency*) es aquella que representa la reutilización de un paquete en el entorno de producción. Aquel en el que se ejecutará la aplicación en el cliente. Este tipo de dependencias se debe registrar en la propiedad dependencies del archivo package.json del paquete.

He aquí un ejemplo ilustrativo:

```
"dependencies": {
  "justo": ">= 0.18.2",
  "lodash": "4.16.4"
}
```

Cada vez que un usuario instale el paquete, npm instalará automáticamente todos los paquetes indicados como dependencia de producción.

Durante el desarrollo, si deseamos añadir una nueva dependencia de producción al archivo package.json, podemos hacerlo básicamente de dos maneras. Manualmente, editando el archivo y añadiéndola a la propiedad dependencies. O bien mediante el comando npm install con la opción --save:

```
npm install --save paquete paquete…
```

Cuando se usa la opción, el comando instalará el paquete y añadirá una entrada en la propiedad dependencies.

Dependencias de desarrollo

En cambio, una dependencia de desarrollo (*development dependency*) es aquella que representa la reutilización de un paquete que sólo se usa durante el desarrollo. Observe la diferencia. Las de producción se instalarán siempre; las de desarrollo sólo cuando estemos desarrollando. Se registran en la propiedad devDependencies del archivo package.json.

Ejemplo:

```
"devDependencies": {
  "justo": ">= 0.18.2",
  "justo-assert": "*"
}
```

Es posible añadir una dependencia de desarrollo mediante npm install, pero hay que usar la opción --save-dev:

```
npm install --save-dev paquete paquete…
```

Esta vez, npm install también instalará el paquete localmente, pero la dependencia del paquete la añadirá a la propiedad devDependencies.

Versiones de las dependencias

Cada dependencia se indica mediante una propiedad. La clave indica el nombre del paquete, mientras que el valor indica su versión o dónde se puede encontrar el paquete.

Cuando se indica una versión, se puede utilizar cualquiera de las siguientes sintaxis:

- versión. Exactamente hay que instalar la versión indicada como, por ejemplo, 0.18.2.
- major.minor.x. La versión major.minor en cualquiera de sus *patches*.
- *. Cualquier versión.
- >versión. Se necesita una versión superior a la indicada. Ejemplo: >0.18.2.
- >=versión. Se necesita la versión indicada o una superior.
- <versión. Una versión anterior a la indicada.
- <=versión. Una versión igual o anterior a la indicada.
- ~versión. Una versión equivalente a la indicada.
- ^versión. Una versión compatible con la indicada.
- versión – version. Una versión entre las indicadas, ambas inclusive.

- http://url. La versión a instalar se encuentra en el URL indicado.

  El recurso indicado debe ser un archivo .tgz.
- https://url. La versión a instalar se encuentra en el URL indicado.

  El recurso indicado debe ser un archivo .tgz.
- git://url. La versión a instalar se encuentra en el repositorio Git indicado.
- file:ruta. La versión a instalar se encuentra en la ruta del disco indicada. La ruta puede ser absoluta o relativa al directorio del archivo package.json.

  Puede ser un directorio o un archivo .tgz.

Instalación de dependencias

Cuando instalamos un paquete mediante npm install, automáticamente se instalan todas sus dependencias. Pero hay ocasiones, durante el desarrollo principalmente, en las que creamos una copia del paquete de desarrollo y deseamos instalar sus dependencias. Por ejemplo, suponga que ha desarrollado una aplicación y desea instalarla para su uso. Podría copiar el contenido del paquete en una ubicación para realizar pruebas. En este caso, ¿cómo se instalan las dependencias de la aplicación? Teniendo en cuenta que las tenemos en el archivo package.json, se puede usar la siguiente sintaxis del comando npm install:

```
npm install
npm install --production
```

Si indicamos --production, sólo instalará las dependencias registradas en la propiedad dependencies. En cambio, si omitimos la opción, instalará todas, tanto dependencies como devDependencies. También es posible instalar *sólo* las dependencias de producción sin usar la opción --production, para ello, hay que fijar la variable de entorno NODE_ENV a production o bien fijar el parámetro de configuración production a true.

Siempre que ejecute una aplicación Node en producción, fije la variable de entorno NODE_ENV a production. Así, sólo se instalarán las dependencias de producción. Generalmente, reduce drásticamente el proceso de instalación. En la mayoría de los proyectos, las dependencias de desarrollo son importantes e instalarlas innecesariamente no tiene sentido. Por otra parte, los paquetes pueden usar esta variable de entorno para hacer determinadas cosas. Por ejemplo, en depuración suele mostrarse mensajes de error más detallados que en producción.

Es posible instalar un determinado paquete localmente, sin necesidad de que se encuentre entre sus dependencias, mediante el comando npm install:

```
npm install paquete
npm install paquete@versión
npm install directorio-paquete
```

Módulo principal o punto de entrada del paquete

En un paquete, cada archivo de código JavaScript se conoce formalmente como módulo (*module*). Ya lo hemos visto en una lección anterior. Se recomienda que cada módulo implemente una determinada funcionalidad del paquete. Clara y bien definida. Por lo general, cada clase de JavaScript se implementa mediante un módulo homónimo.

El módulo principal (*main module*) o punto de entrada (*entry point*) es aquel que ejecuta node cuando importa por primera vez el paquete. Por convenio y buenas prácticas, se usa el archivo index.js. Y se indica mediante la propiedad main del archivo package.json. Si seguimos el convenio, tendremos:

```
"main": "index.js"
```

Cualquier otro módulo se considera un módulo secundario (*secondary module*). Y por convenio, se suelen encontrar en la carpeta lib del paquete.

API del paquete

Los paquetes, al igual que los módulos, presentan una API, recordemos, un conjunto de objetos que ofrece el paquete para su reutilización. La API de un paquete es la API de su módulo principal. Lo que exporte este módulo será lo que los usuarios del paquete podrán reutilizar.

Archivo .npmignore

El archivo .npmignore enumera los archivos y/o directorios del proyecto que no forman parte del paquete. Nunca se publicarán en el repositorio de NPM.

Su sintaxis es la misma que la de .gitignore. Los comentarios son aquellas líneas que comienzan por almohadilla (#). Cada archivo o directorio se puede indicar mediante una ruta relativa al directorio del proyecto. Las rutas se pueden indicar mediante patrones. Cuando una ruta finaliza en /, indica que se trata de un directorio e incluye todo su contenido. Si la ruta se precede de un signo de exclamación (!), niega el patrón que le sigue.

Aunque no se incluyan en .npmignore, siempre se excluye las siguientes entradas:

- .git
- .npmrc
- .npm-debug.log
- node_modules/

Aunque se incluyan en .npmignore, *nunca* se excluye las siguientes entradas:

- package.json
- README en cualquiera de sus variantes.
- CHANGELOG en cualquiera de sus variantes.
- LICENSE
- LICENCE

npm link

En desarrollo, es muy útil el comando npm link. Crea un enlace *global* al directorio de un paquete:

```
npm link
npm link directorio-del-paquete
```

Crea una entrada en el directorio predeterminado del usuario que no es más que un enlace al directorio actual o al indicado. El directorio enlazado debe contener una instalación correcta del paquete, incluidos los paquetes de los que depende.

La idea es simular que está instalado, pero sin estarlo. De esta manera, podemos probar un paquete durante su desarrollo sin necesidad de publicarlo e instalarlo, ni reinstalarlo una y otra vez a lo largo del desarrollo. Cualquier cambio en el paquete no requiere su reinstalación, gracias a que se está usando el enlace global.

Paquetes con ámbito

Ya sabemos que un paquete es un componente de software reutilizable. Puede ser reutilizado en un único proyecto o en varios. La cuestión es que implementa una funcionalidad clara y bien definida, que al desarrollarse mediante un paquete es más fácil de mantener y probar.

NPM permite lo que se conoce como paquetes con ámbito (*scoped packages*), aquellos cuyo nombre de paquete tiene un formato como el siguiente:

```
@ámbito/paquete
```

El ámbito (*scope*) no es más que un espacio de nombres dentro del cual se ubica el paquete. Se utiliza para ubicar dentro del mismo ámbito una colección de paquetes relacionados. En el momento de su instalación, todos los paquetes con el mismo ámbito se instalarán dentro de la misma carpeta. Así por ejemplo si tenemos los paquetes @a/p y @a/q, la instalación se realizará en node_modules/@a/p y node_modules/@a/q.

Su importación es similar a la presentada hasta ahora, lo único que tendremos que indicar tanto el ámbito como el nombre del paquete. Ejemplo:

```
const pkg = require("@scope/pkg");
import pkg from "@scope/pkg";
```

