¡Les damos la bienvenida!

¿Comenzamos?

Esta clase va a ser

- grabada

Clase 04. DESARROLLO AVANZADO DE BACKEND

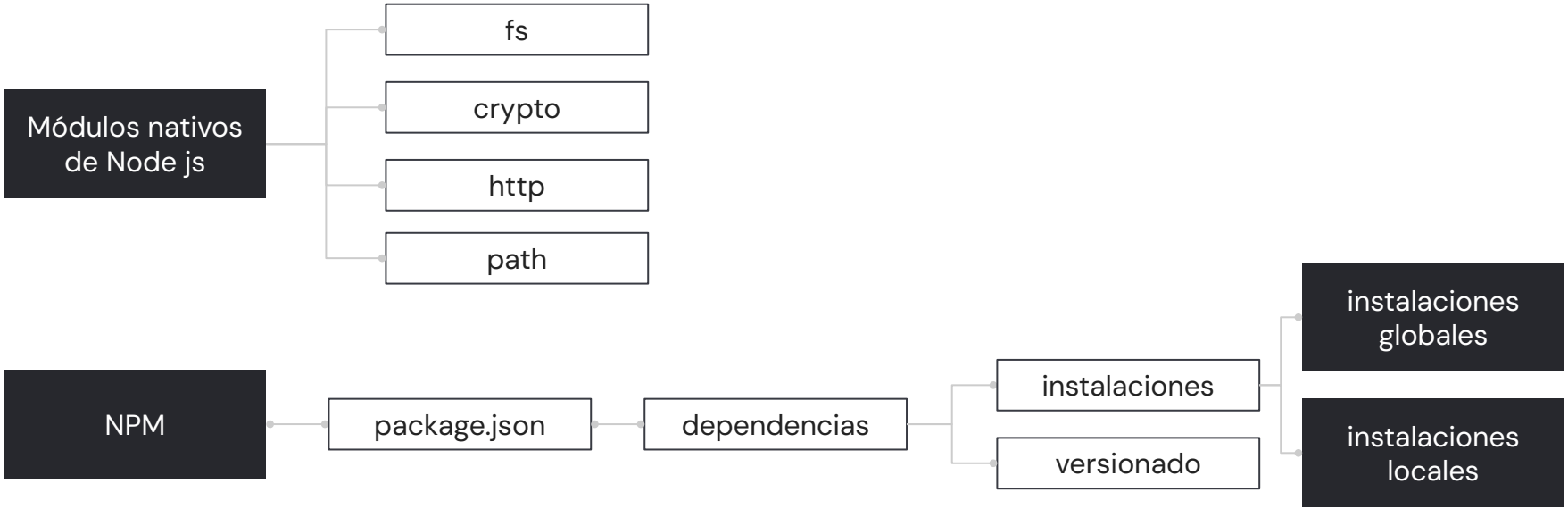
Administrador de Paquetes – NPM

CODERHOUSE

Objetivos de la clase

- Repasar qué es Node js y su uso en el backend
- Entender la diferencia entre un módulo nativo y uno de terceros
- Conocer la función de NPM y el proceso de instalación de dependencias
- Conocer el proceso de actualización de dependencias.

MAPA DE CONCEPTOS



Repasando nuestra herramienta de trabajo: Node Js

Node js

Surgió de la necesidad de ejecutar javascript fuera del navegador, y ha crecido hasta convertirse en uno de los elementos principales para el desarrollo web.

Cuenta con el mismo motor V8 de Google Chrome, el cual permite convertir el código javascript a código máquina para poder ser procesado correctamente.

Además, cuenta con muchas funcionalidades internas del mismo lenguaje javascript gracias a sus ajustes con ECMAScript.

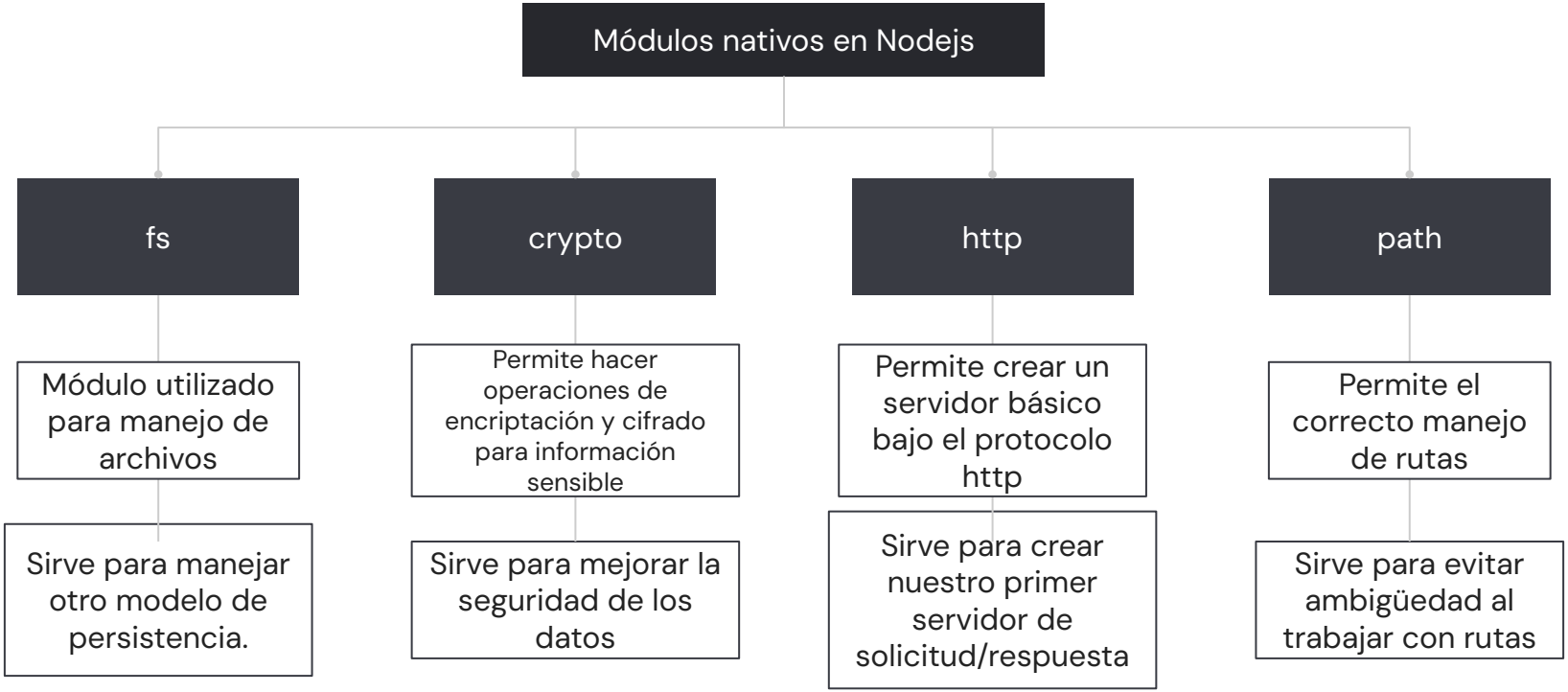
ACTIVIDAD EN CLASE

Proyecto de node

- ✓ Crear un proyecto de node que genere 10000 números aleatorios en un rango de 1 a 20. Indicar por consola la finalización de esta operación con un mensaje.
- ✓ Mediante el uso de Promesas, crear un objeto cuyas claves sean los números salidos y el valor asociado a cada clave será la cantidad de veces que salió dicho número. Representar por consola los resultados.

Nota: Considerar que esta operación debe realizarse de forma asíncrona.

Módulos nativos de Node js


Módulos nativos en Nodejs
fs
crypto
http
path
Módulo utilizado para manejo de archivos
Permite hacer operaciones de encriptación y cifrado para información sensible
Permite crear un servidor básico bajo el protocolo http
Permite el correcto manejo de rutas
Sirve para manejar otro modelo de persistencia.
Sirve para mejorar la seguridad de los datos
Sirve para crear nuestro primer servidor de solicitud/respuesta
Sirve para evitar ambigüedad al trabajar con rutas

¡Importante!

Recuerda que utilizamos soluciones de terceros para hacer nuestro trabajo mejor. **Utilizar un módulo que nos permita solucionar un problema previo, permite concentrarnos en el problema actual.**

HANDS ON LAB

Práctica de módulo nativo: crypto

¿Cómo lo hacemos? **Se creará una clase “UsersManager” que permitirá guardar usuarios en un atributo estático. El usuario se recibirá con una contraseña en string plano, y se deberá guardar la contraseña hasheada con crypto. Utilizar el módulo nativo crypto.**
El manager debe contar con los siguientes métodos:

- ✓ El método “Crear usuario” debe recibir un objeto con los campos:
  - Nombre
  - Apellido
  - Nombre de usuario
  - Contraseña

  El método debe guardar un usuario en un atributo estático llamado “Usuarios”, **recordando que la contraseña debe estar hasheada por seguridad**

HANDS ON LAB

Práctica de módulo nativo: crypto

- ✓ El método “Mostrar Usuarios” imprimirá en consola todos los usuarios almacenados.
- ✓ El método “Validar Usuario” recibirá el nombre de usuario que quiero validar, seguido de la contraseña, debe poder leer el json previamente generado con el arreglo de usuarios y hacer la comparación de contraseñas, Si coinciden el usuario y la contraseña, devolver un mensaje “Logueado”, caso contrario indicar error si el usuario no existe, o si la contraseña no coincide.

Manejando módulos de terceros: NPM

¿Qué es NPM?

NPM refiere a las siglas “Node Package Manager”, El cual refiere a un manejador de paquetes de Node. Éste permite que la comunidad de desarrolladores puedan **crear sus propios módulos**, para poder subirlos a la nube y así otros desarrolladores puedan utilizarlos.

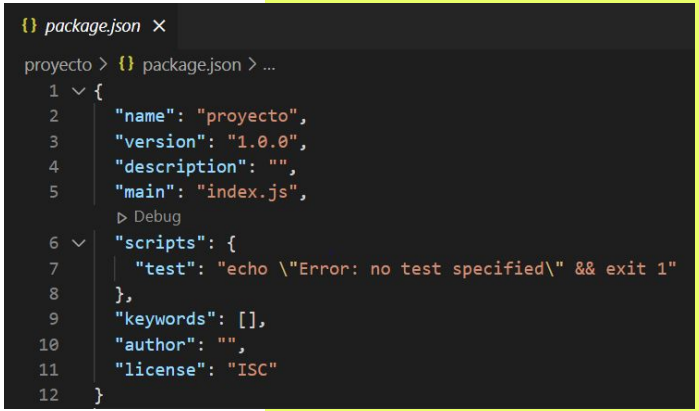
Para el trabajo de paqueterías, tendremos un archivo en nuestro proyecto llamado **package.json**

package.json

¿Qué es package.json?

package.json es un archivo que generamos dentro de nuestros proyectos, el cual contendrá distintas especificaciones del mismo, cosas como:

- El nombre de tu proyecto
- La versión de tu proyecto
- Algunos scripts para correr el proyecto
- **¿de qué depende el proyecto?**



CODERHOUSE

Dependencias

Cuando nuestro proyecto necesita utilizar dependencias de terceros a partir de npm, se añade un nuevo campo a nuestro package.json llamado “dependencies” el cual contendrá los módulos que tenemos instalados en ese proyecto y, por lo tanto, indica que **el proyecto necesita de esas dependencias instaladas para poder correr correctamente**.

Instalando nuestra primera dependencia

Break

¡10 minutos y volvemos!

Para pensar

- ✓ Si los módulos que yo instalo son de terceras personas, ¿cómo saber qué puedo hacer con ellos y qué no?

- ✓ ¿Debo hacer la instalación del módulo en cada proyecto que vaya a hacer?

Instalaciones globales e instalaciones locales

¿Global o local?

Instalar una dependencia **de manera local** significa que ese módulo instalado pertenecerá y se utilizará **sólo dentro de ese proyecto**. Ello implica que, si quisiera utilizar la misma dependencia en otro proyecto, tendría que volver a hacer la instalación, ya que **no son compartidas**.

```
Ejemplo> npm install modulo_a_instalar
```

Por otra parte, instalar una dependencia **de manera global** implica instalar el módulo **para todos los proyectos**, evitando la necesidad de instalar cada vez que hagamos un proyecto nuevo. Para instalar de manera global, sólo colocamos la flag **-g**

```
Ejemplo> npm install -g modulo_a_instalar
```

Versionado de dependencias

Manejo de versiones en NPM

Las versiones se basan en 3 elementos básicos:

- ✓ Versiones mayores (primer dígito): Hace referencia a cambios **grandes, tanto que ya no son compatibles con otras versiones anteriores**.
- ✓ Versiones menores (segundo dígito): Hace referencia a cambios en ciertas características y funcionalidades **que no afecten a versiones anteriores**, es decir, podemos actualizarlo sin afectar la estructura del proyecto. Símbolo de actualización: **^**
- ✓ Parches (último dígito): Hace referencia a bugfixes o manejo de defectos del código actual. **No se está cambiando nada estructuralmente hablando**, sólo estamos arreglando cosas. Símbolo de actualización: **~**

Política de actualizaciones y dependencias

Comandos para actualizar en NPM

`npm outdated` es un comando que leerá las dependencias instaladas en nuestro package.json y, **según el operador que hayamos colocado**, nos indicará qué es lo que nos "conviene". También nos indica cuál es la última versión encontrada en internet, en caso de que nos interese.

```
ecommerce_backend> npm outdated
Package    Current   Wanted    Latest  Location                 Depended by
aws-sdk    2.1122.0  2.1128.0  2.1128.0  node_modules/aws-sdk   ecommerce_backend
express      4.18.0    4.18.1    4.18.1  node_modules/express   ecommerce_backend
mongoose      6.3.1     6.3.2     6.3.2  node_modules/mongoose  ecommerce_backend
```

Para poder llevar a cabo la actualización, utilizaremos el comando `npm update` el cual se encargará de realizar los cambios que indicamos

ACTIVIDAD EN CLASE

Calculadora de edad

Realizar un programa que utilice la dependencia **momentjs** (deberá instalarse por npm install).

- ✓ Debe contar con una variable que almacene la fecha actual (utilizar moment())
- ✓ Debe contar con una variable que almacene sólo la fecha de tu nacimiento (utilizar moment).
- ✓ Validar con un if que la variable contenga una fecha válida (utilizar el método isValid());
- ✓ Finalmente, mostrar por consola cuántos días han pasado desde que naciste hasta el día de hoy. (utilizar el método diff()
- ✓ Extra: Cambia tu moment a la versión 1.6.0, al no ser la misma versión mayor, nota el cambio al correr el programa.

CODERHOUSE

¿Preguntas?

Opina y valora esta clase

Muchas gracias.