



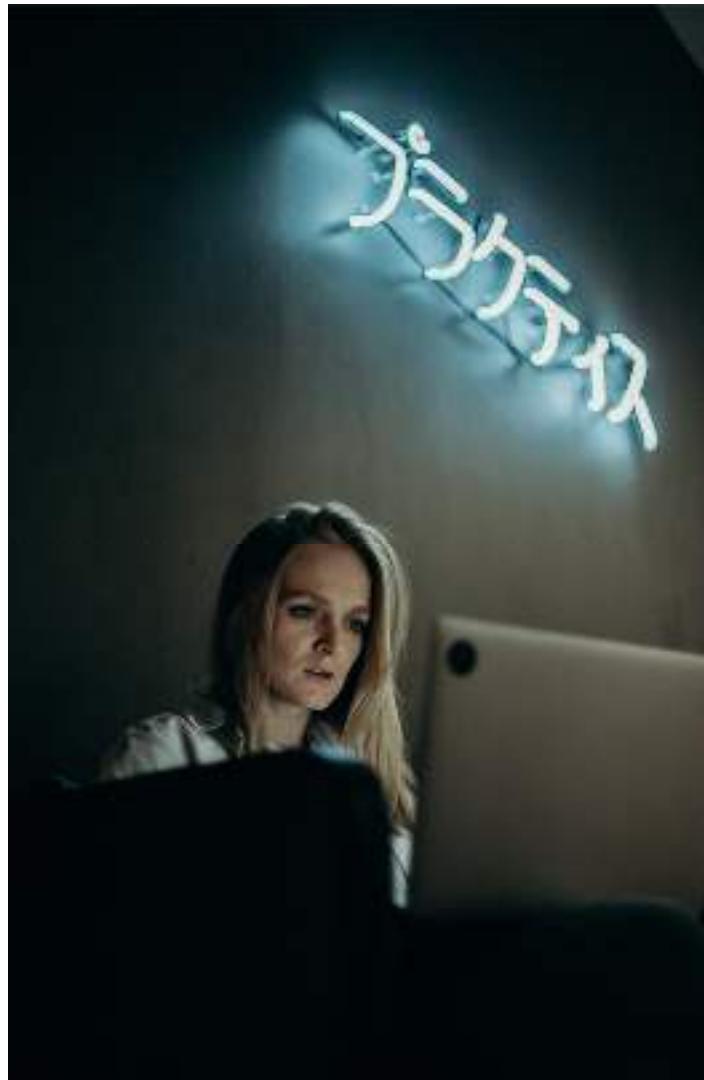
¡Les damos la bienvenida!

¿Comenzamos?

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE



COMISIÓN N°#####

Presentación del equipo

- ✓ Profesor responsable: Juan Pérez
- ✓ Coordinador: Juan Pérez
- ✓ Tutores:

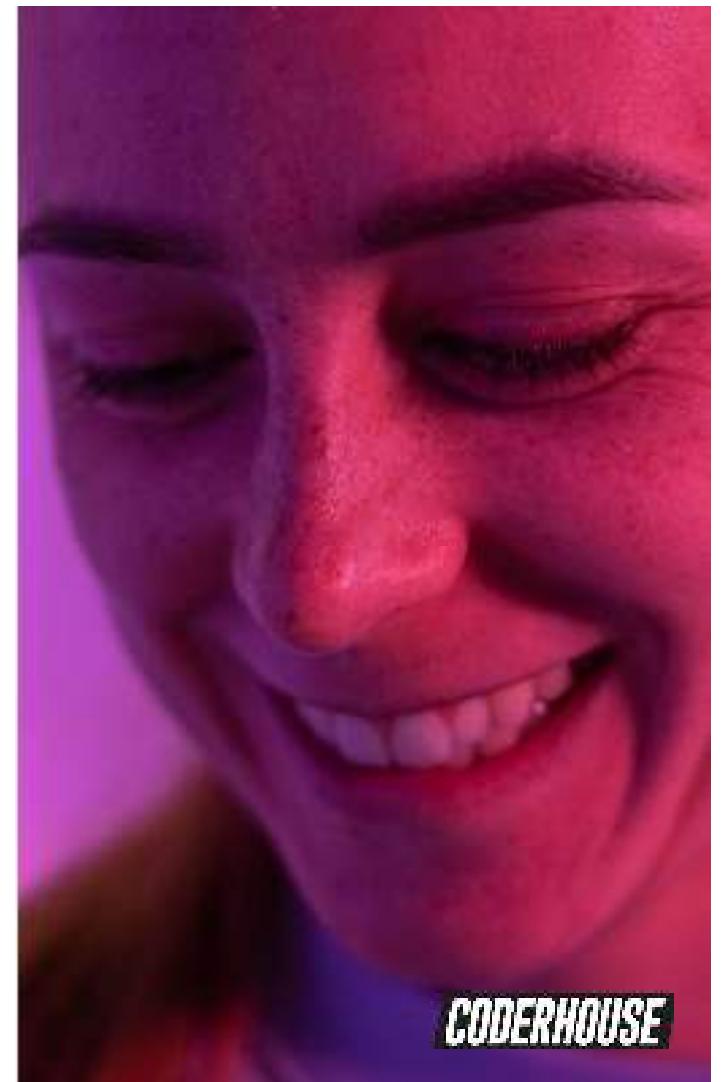
- | | |
|--------------|--------------|
| ○ Juan Pérez | ○ Juan Pérez |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |

CODERHOUSE

Presentación de estudiantes

Por encuestas de Zoom

1. País
2. Conocimientos previos
3. ¿Por qué elegiste este curso?



¿Dudas sobre el onboarding?

Míralo aquí

CODERHOUSE

Clase 0. Testing y Escalabilidad Backend

Proceso principal del servidor + Global & Child process

CODERHOUSE

Temario

0

Proceso principal del servidor + Global & Child process

- ✓ Process
- ✓ Manejo de variables de entorno
- ✓ Global & child process

01

Testing y Mocks

- ✓ TDD
- ✓ Desarrollo de prácticas preventivas a partir de mocks

CODERHOUSE

Objetivos de la clase

- **Conocer** el objeto process en nodejs
- **Entender** las implementaciones principales de un process
- **Aprender a manejar** variables de entornos

Process

CODERHOUSE

¿Qué es process?

Cada vez que corremos un proceso en nodejs, éste genera un objeto llamado **process**, el cual contiene información referente a todo lo implicado con el proceso, cosas como:

- ✓ Uso de memoria
- ✓ Id del proceso en el sistema operativo
- ✓ En qué sistema operativo o plataforma está corriendo
- ✓ En qué entorno está corriendo.
- ✓ Qué argumentos tiene el entorno.

Algunos elementos importantes de process

- ✓ `process.cwd()` : Directorio actual del proceso.
- ✓ `process.pid` : id del proceso en el sistema
- ✓ `process.MemoryUsage()` :
- ✓ `process.env` : Accede al objeto del entorno actual
- ✓ `process.argv` : Muestra los argumentos pasados por CLI
- ✓ `process.version` : Muestra la versión del proceso (node en este caso)
- ✓ `process.on()` : Permite setear un listener de eventos
- ✓ `process.exit()` : Permite salir del proceso.

Manejo de Argumentos

CODERHOUSE

Argumentos en consola

Los argumentos permiten iniciar la ejecución de un programa a partir de ciertos elementos iniciales. Con argumentos podemos:

- ✓ Setear configuraciones de arranque
- ✓ Agregar valores predeterminados
- ✓ Resolver outputs específicos

Para poder trabajar con argumentos, podemos hacerlo a partir de **`process.argv`**.

💡 Recordemos que, por defecto, `process.argv` siempre tendrá dos elementos iniciales

Ejecutando process con diferentes argumentos

```
JS process.js X  
  
JS process.js  
1 //En todos los ejemplos retiramos los dos argumentos por default  
2  
3 //Si ejecutamos " node process.js 1 2 3"  
4 console.log(process.argv.slice(2)) //Imprimirá [1,2,3]  
5  
6 //Si ejecutamos " node process.js a 2 -a"  
7 console.log(process.argv.slice(2)) //Imprimirá ['a', '2', '-a']  
8  
9 //Si ejecutamos " node process.js "  
10 console.log(process.argv.slice(2)) //Imprimirá []  
11  
12 //Si ejecutamos " node process.js --mode development"  
13 console.log(process.argv.slice(2)) //Imprimirá ['--mode', 'development']
```

Procesamiento de argumentos con Commander

CODERHOUSE

Commander

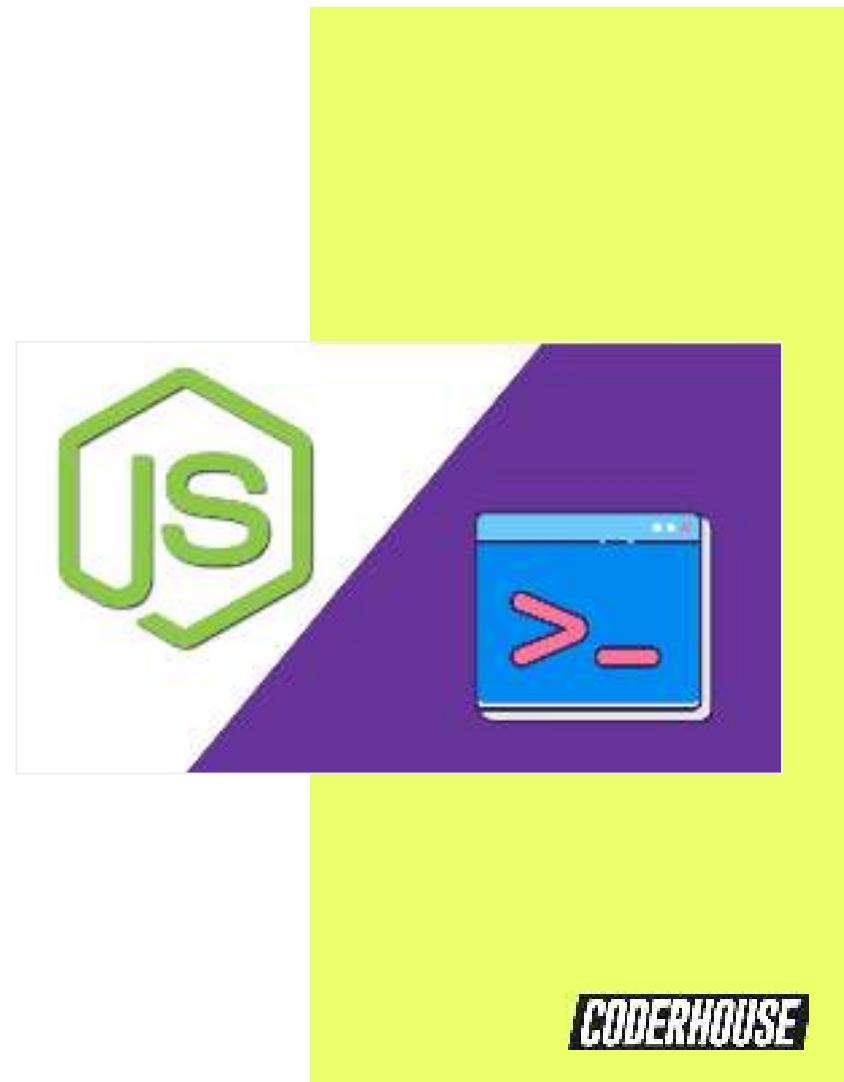
Commander es una de las librerías más utilizadas a la fecha para el manejo de argumentos. Permite realizar funciones como:

- ✓ Convertir flags directamente en booleanos
- ✓ Limitar sólo las flags configuradas (cualquier otra impide el procesamiento del programa)
- ✓ Colocar argumentos predeterminados

Y muchas herramientas más para trabajar con argumentos!

Instalamos commander con el comando

```
npm install commander
```



CODERHOUSE

Ejemplo de uso de commander

```
1< process> X {# package.json
2< process> >-
3  import {Command} from 'commander';
4
5  const program = new Command(); // Inicializamos un nuevo comando do commander
6
7  program
8    .option('-d', 'Variable para debug', false) // primero: el comando, segundo: descripción,
// tercero: valor default
9    .option('-p <port>', 'puerto del servidor', 8080) // port es el puerto a colocar
10   .option('--mode <mode>', 'Modo de trabajo', 'production') // mode es el argumento a colocar
11   .requiredOption('-u <user>', 'Usuário utilizando el aplicativo', 'No se ha declarado un usuário') // para requiredOption, el tercer arg es un
// mensaje de error en caso de que no se especifique
12   .option('-l', '--letters [letters...]') // specify letters
13
14  program.parse(); //parse se utiliza para cerrar la configuración de comandos.
15
16  console.log('Options: ', program.opts());
17  console.log('Remaining arguments: ', program.args);
```

Enviando argumentos a commander

```
node .\process.js -d -p 3000 --mode development -u root --letters a b c
```

```
Options: {  
  d: true,  
  p: '3000',  
  mode: 'development',  
  u: 'root',  
  letters: [ 'a', 'b', 'c' ]  
}  
Remaining arguments: []
```

```
node .\process.js -p 3000 -u root 2 a 5 --letters a b c
```

```
Options: {  
  d: false,  
  p: '3000',  
  mode: 'production',  
  u: 'root',  
  letters: [ 'a', 'b', 'c' ]  
}  
Remaining arguments: [ '2', 'a', '5' ]
```



Para pensar

El manejo de argumentos permite iniciar un programa con valores iniciales

¿Con qué fines consideras que puede utilizarse?

Manejo de variables de entorno

CODERHOUSE



Sobre los entornos

Seguramente, a estas alturas del desarrollo entenderás que un código no puede simplemente hacerse y llegar al cliente.

Para que un código esté listo para llegar al cliente, es necesario que pase por diferentes fases.

Sin embargo, para que estas fases se encuentren aisladas de las otras fases (no queremos que la fase de desarrollo tenga datos de producción, o que haya datos de producción en staging), necesitaremos crear entornos específicos para estas fases.

Nuestras variables cambiarán según el entorno.

Usos de una variable de entorno

El primer uso radica en que una variable cambie según el entorno donde se esté corriendo, esto permite que pueda apuntar a una base de datos prueba o a una base de datos productiva con sólo cambiar el apuntador de dónde se está corriendo.

Otro factor importante es el factor seguridad. Con las variables de entorno podemos ocultar la información sensible de nuestro código, como credenciales, claves de acceso, tokens, secrets, etc.

dotenv

CODERHOUSE

Utilizando dotenv

La librería dotenv nos permitirá setear un entorno en un archivo `.env`, a partir de éste, colocaremos todas las variables que queramos proteger y mantener de manera dinámica.

El archivo no requiere ningún tipo de consideración, bastará con declarar variables de la forma

NOMBRE = VALOR

Para comenzar con dotenv, basta hacer

```
npm install dotenv
```



CODERHOUSE

Archivo config

Una vez instalado dotenv, bastará contar con un archivo de configuración llamado **config.js**. En éste, colocaremos un objeto, donde cada key : value corresponderá a la variable de dotenv. todo se basará en utilizar el objeto process nuevamente, esta vez usando

process.env.VARIABLE

Usando **dotenv.config()**, indicamos a la computadora que cargue las variables del archivo **.env**, el cual estará seteado dentro del proyecto.

```
JS config.js  X
src > config > JS config.js > set default > adminPassword
1 import dotenv from 'dotenv';
2
3 dotenv.config();
4
5
6 export default {
7   port:process.env.PORT,
8   mongoUrl:process.env.MONGO_URI,
9   adminName : process.env.ADMIN_NAME,
10  adminPassword : process.env.ADMIN_PASSWORD
11 }
```

```
env  X  JS config.js
.env
1 PORT = 8888
2 MONGO_URL = mongodb://127.0.0.1:27017/base
3 ADMIN_NAME = adminCoder
4 ADMIN_PASSWORD = AdMinC0d3R2022
```

Utilizando la configuración

```
src > ls index.js
1   import config from './config/config.js';
2
3
4   console.log(config);
```

```
{
  port: '8080',
  mongoUrl: 'mongodb://127.0.0.1:27017/base',
  adminName: 'adminCoder',
  adminPassword: 'AdM1nC0d3R2022'
}
```

Una vez finalizada la configuración, podemos importarla y visualizar los resultados:

Al final lo que ocurre es lo siguiente: al ejecutar el comando dotenv.config(), éste busca las variables localizadas en .env y procede a colocarlas en el objeto **process.env**.

¡Entonces conseguimos tener un objeto de configuración listo para este entorno!

Múltiples entornos

CODERHOUSE

Múltiples entornos = múltiples archivos

Al trabajar con más de un entorno (por ejemplo, desarrollo Y producción), es necesario setear diferentes archivos con el fin de apuntar a diferentes elementos (Por ejemplo, el entorno de desarrollo podría almacenar en archivos, y el entorno productivo podría almacenar en base de datos).

Para esto, setearemos un archivo por cada entorno que deseamos trabajar.

La idea es que ambos archivos tengan las mismas variables, pero con diferentes valores.



```
.env.development
1. PORT = 8080
2. MONGO_URL = mongodb://127.0.0.1:27017
3. ADMINT_NAME = adminCoder
4. ADMINT_PASSWORD = AdminCoder123

.env.production
1. PORT = 3000
2. MONGO_URL = mongodb://127.0.0.1:27028,
3. ADMINT_NAME = adminCoderProductivo
4. ADMINT_PASSWORD = PappaConequito
```

.env.development
.env.production

Luego, en dotenv...

La ejecución de dotenv.config() ya no bastará, ya que esta vez tenemos que poder identificar a cuál archivo queremos apuntar. Para esto utilizaremos la configuración **path**, apuntaremos según lo indicado por la constante "environment".



```
JS config.js  X  JS index.js
src > config > JS config.js > [x] environment
1 import dotenv from 'dotenv';
2
3 const environment = "PRODUCTION";
4 dotenv.config({
5   path: environment === "DEVELOPMENT" ? './.env.development' : './.env.production'
6 });
7
```

Ejemplo de diferencias según cómo se manda a llamar al entorno

Ejecutando con environment =
DEVELOPMENT

```
{  
  port: '8080',  
  mongoUrl: 'mongodb://127.0.0.1:27017/base',  
  adminName: 'adminCoder',  
  adminPassword: 'AdMinCod3R2022'}
```

Ejecutando con environment =
PRODUCTION

```
{  
  port: '3000',  
  mongoUrl: 'mongodb://127.0.0.1:27020/base',  
  adminName: 'adminCoderProductive',  
  adminPassword: 'PapaConQuesito'}
```



Utilizando argumentos con dotenv

Duración: 10 min

CODERHOUSE



ACTIVIDAD EN CLASE

Utilizando argumentos con dotenv

Consigna:

- ✓ Realizar un servidor basado en node js con express, El cual reciba por flag de cli el comando **--mode <modo>** y sea procesado por commander.
- ✓ Acorde con este argumento, hacer una lectura a los diferentes entornos, y ejecutar dotenv en el path correspondiente a cada modo (**--mode development** debería conectar con **.env.development**).
- ✓ Para el entorno **development**, el servidor debe escuchar en el puerto 8080, para el entorno productivo, el servidor debe escuchar en el puerto 3000.



Break

¡10 minutos y volvemos!

CODERHOUSE

Continuando con
process
Global & child process

CODERHOUSE

Listeners

Además de todas las funcionalidades que hemos visto de process, tenemos otra más para trabajar: el método **on**. Permitirá poner a nuestro proceso principal a la escucha de algún evento para poder ejecutar alguna acción en caso de que algo ocurra.

Algunos de los listeners más utilizados son

- ✓ on 'exit' : Para ejecutar un código **justo antes** de la finalización del proceso.
- ✓ on 'uncaughtException' : Para atrapar alguna excepción que no haya sido considerada en algún catch
- ✓ on 'message' para poder comunicarse con otro proceso

listeners con process.on

```
1 process.on('exit',code=>{
2     console.log(`Este código se ejecutará justo antes de salir del proceso`)
3 })
4 process.on('uncaughtException',exception=>{
5     console.log(`Este código atrapa todas las excepciones no controladas
6     como llamar una función que no haya sido declarada`)
7 })
8 process.on('message',message=>{
9     console.log(`Este código se ejecutará cuando reciba un mensaje de otro
10    proceso`)
11 })
```

Por ejemplo, sin en algún punto llegase a cometer algún error, como llamar console() en lugar de console.log()

```
})
//Al no existir la función console(), ocurrirá una excepción NO CONTROLADA
console();
```

Este código atrapa todas las excepciones no controladas
como llamar una función que no haya sido declarada
Este código se ejecutará justo antes de salir del proceso

Códigos de salida de process

Cuando ejecutamos una salida con `process.exit()` como argumento, podemos enviar un código que sirve como identificador para el desarrollador sobre la razón de la salida

Hay que conocer los códigos de salida para saber cómo utilizarlos. También podemos crear nuestros propios códigos.

Algunos de los códigos importantes son:

0 : proceso finalizado normalmente.

1 : proceso finalizado por excepción fatal

5 : Error fatal del motor V8.

9 : Para argumentos inválidos al momento de la ejecución.



Ejemplo en vivo

- ✓ Se creará una función llamada “listNumbers” el cual recibirá un número indefinido de argumentos (...numbers)
- ✓ Si se pasa un argumento no numérico, entonces deberá mostrar por consola un error indicando “Invalid parameters” seguido de una lista con los tipos de dato (para [1,2,”a”,true], el error mostrará [number,number,string,boolean]
- ✓ Escapar del proceso con un código -4. Utilizando un listener, obtener el código de escape del error y mostrar un mensaje “Proceso finalizado por argumentación inválida en una función”

Duración: **15 min**

Child Process

CODERHOUSE

Un proceso creando otro proceso

Existen casos en los que un proceso de node necesitará crear otro proceso para poder resolver una función de gran complejidad.

Algunas operaciones requieren mucho procesamiento, como:

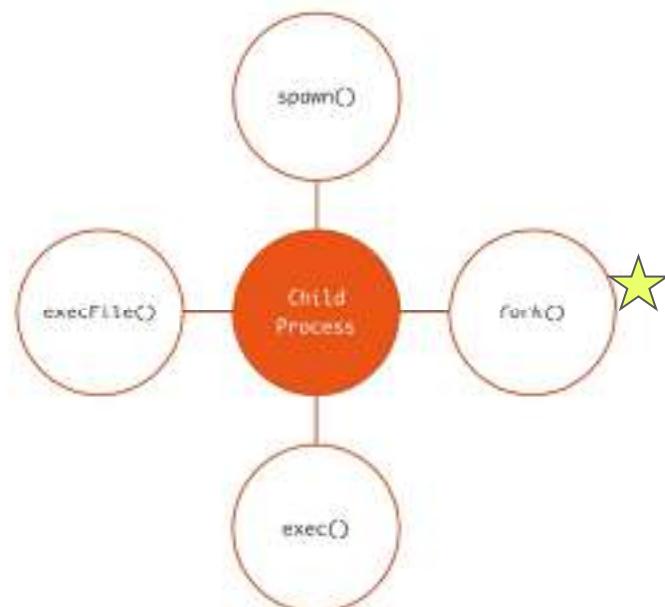
- ✓ Lectura de archivos enormes
- ✓ Consultas a bases muy complejas

Por lo que, para no bloquear las tareas actuales de, un servidor, por ejemplo, ocupamos separar esa tarea en otro subprocesso.

Cómo crear un child process

Existen diferentes formas para que un proceso de node pueda ejecutar otro proceso, hay cuatro operadores que pueden ser utilizados y manipulados de diferentes formas

En esta clase aprenderemos sobre el método fork(). Sin embargo, se te invita a que profundices sobre los diferentes métodos e indagues contextos de aplicación.





APROXIMACIÓN AL PROCESO

método fork: problema

Caso de uso de fork

Supongamos que tenemos una operación en nuestro servidor express que consume muchos recursos y demora una fuerte cantidad de tiempo (en este caso, sólo será una suma).

```
function operacionCompleja(){
  let result = 0;

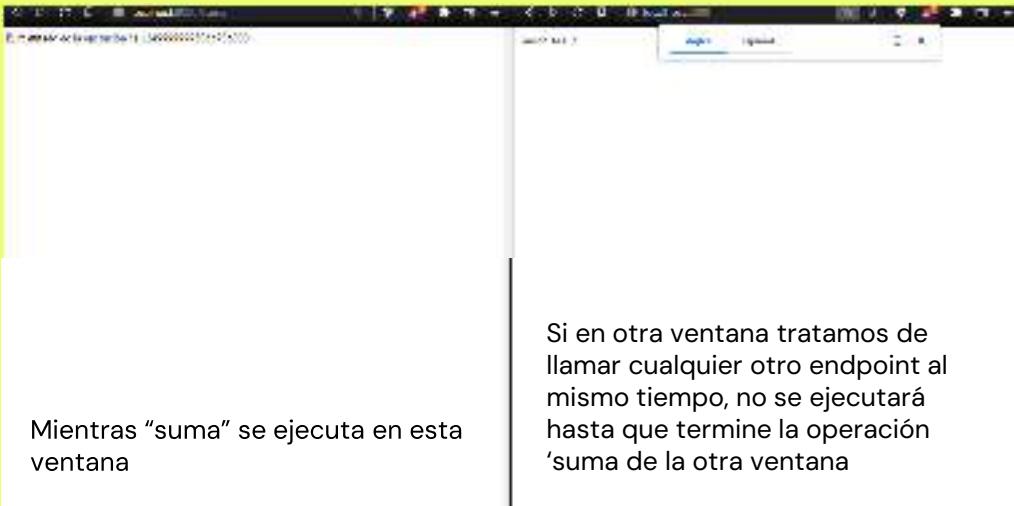
  for(let i = 0;i<5e9;i++){
    result+=i;
  }
  return result;
}

app.get('/suma',(req,res)=>{
  const result = operacionCompleja();
  res.send(`El resultado de la operación es ${result}`);
})
```



APROXIMACIÓN AL PROCESO

método fork: problema



Mientras "suma" se ejecuta en esta ventana

Si en otra ventana tratamos de llamar cualquier otro endpoint al mismo tiempo, no se ejecutará hasta que termine la operación 'suma de la otra ventana

Comienzan las complicaciones

Al llamar al endpoint '/suma', notamos algo horrible! Las operaciones se han bloqueado en todos los demás endpoints si se llaman en paralelo.

Pensando una solución

No podemos simplemente “no hacer la operación”. Debemos buscar la manera de que esta operación se pueda hacer sin bloquear otros endpoints. 🤔

¿Y si delegamos a un proceso hijo para que sólo se encargue de ese proceso, mientras yo sigo atendiendo solicitudes? Suena bien ¿no?

APROXIMACIÓN AL PROCESO

Comenzando a estructurar un forkeo

1. Separar la función que está causando problemas en un archivo diferente y reestructurarla para que sólo se ejecute cuando el padre se lo indique (usaremos el process.on('message'))

```
js operacionCompleja.js ×
src > JS operacionCompleja.js > ⚡ operacionCompleja
1 <--> function operacionCompleja(){
2     let result = 0;
3
4     for(let i = 0;i<5e9;i++){
5         result+=i;
6     }
7     return result;
8 }
```

```
js operacionCompleja.js ×
src > JS operacionCompleja.js > ⚡ process.on('message') callback
1 <--> process.on('message',message=>{
2     let result = 0;
3
4     for(let i = 0;i<5e9;i++){
5         result+=i;
6     }
7     process.send(result);
8 })
```

el return se convierte en process.send

CODERHOUSE

APROXIMACIÓN AL PROCESO

Volviendo a nuestro app.js

Comenzamos importando fork:

```
import fork from 'child_process';
```

No es necesario instalar nada, el módulo “child_process” ya es un proceso nativo. Luego. Reestructuramos nuestro endpoint /suma

```
app.get('/suma',(req,res)=>{
  const child = fork('./operacionCompleja.js'); //Primero forkeamos la operación
  child.send('¡Inicia el cálculo, por favor!'); //El padre envía un mensaje al hijo
  child.on('message',result=>{ //Sólo hasta que el hijo nos responda, procedemos a mostrar el resultado
    res.send(`El resultado de la operación es ${result}`);
  })
})
```

Recapitulando...

- El padre realiza un **fork** al proceso hijo.
- El padre **envía un mensaje** al proceso hijo
- El proceso hijo **tiene su propio listener**, al recibir el mensaje del padre, entiende que tiene que comenzar con su cálculo.
- Una vez que el hijo termina de calcular, le **reenvía un mensaje** al padre, donde el contenido del mensaje será el resultado.
- Ese resultado se envía al cliente.



Cálculo bloqueante con contador

Duración: 15 min

CODERHOUSE



ACTIVIDAD EN CLASE

Cálculo bloqueante con contador

Realizar un servidor en express que contenga una ruta raíz '/' donde se represente la cantidad de visitas totales a este endpoint

Se implementará otra ruta '/calculo-bloq', que permita realizar una suma incremental de los números del 0 al 100000 con el siguiente algoritmo.

```
function sumar() {  
  let suma = 0  
  for(let i=0; i<5e9; i++) {  
    suma += i  
  }  
  return suma  
}
```



ACTIVIDAD EN CLASE

Cálculo bloqueante con contador

Comprobar que al alcanzar esta ruta en una pestaña del navegador, el proceso queda en espera del resultado. Constatar que durante dicha espera, la ruta de visitas no responde hasta terminar este proceso.

Luego crear la ruta '/calculo-nobloq' que hará dicho cálculo forkeando el algoritmo en un child_process, comprobando ahora que el request a esta ruta no bloquee la ruta de visitas.

¿Preguntas?

CODERHOUSE

**Opina y valora
esta clase**

CODERHOUSE

Resumen de la clase hoy

- ✓ Process
- ✓ Manejo de variables de entorno
- ✓ Global & child process

Muchas gracias.

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE



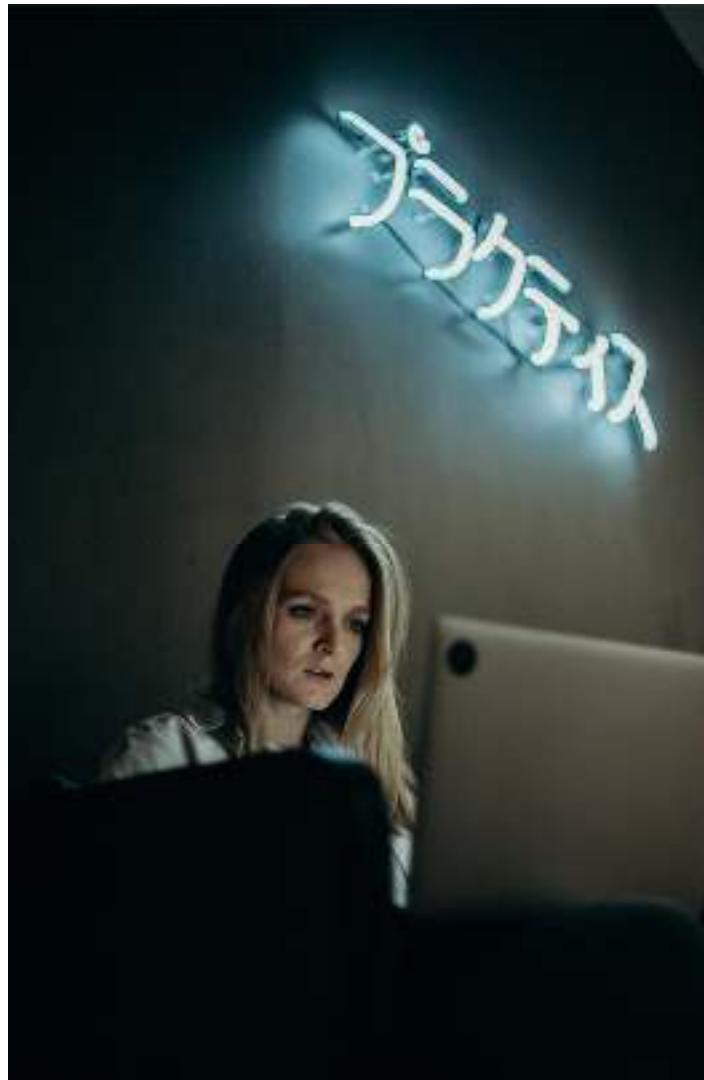
¡Les damos la bienvenida!

¿Comenzamos?

CODERHOUSE

Esta clase va a ser

- **grabada**



COMISIÓN N°#####

Presentación del equipo

- ✓ Profesor responsable: Juan Pérez
- ✓ Coordinador: Juan Pérez
- ✓ Tutores:

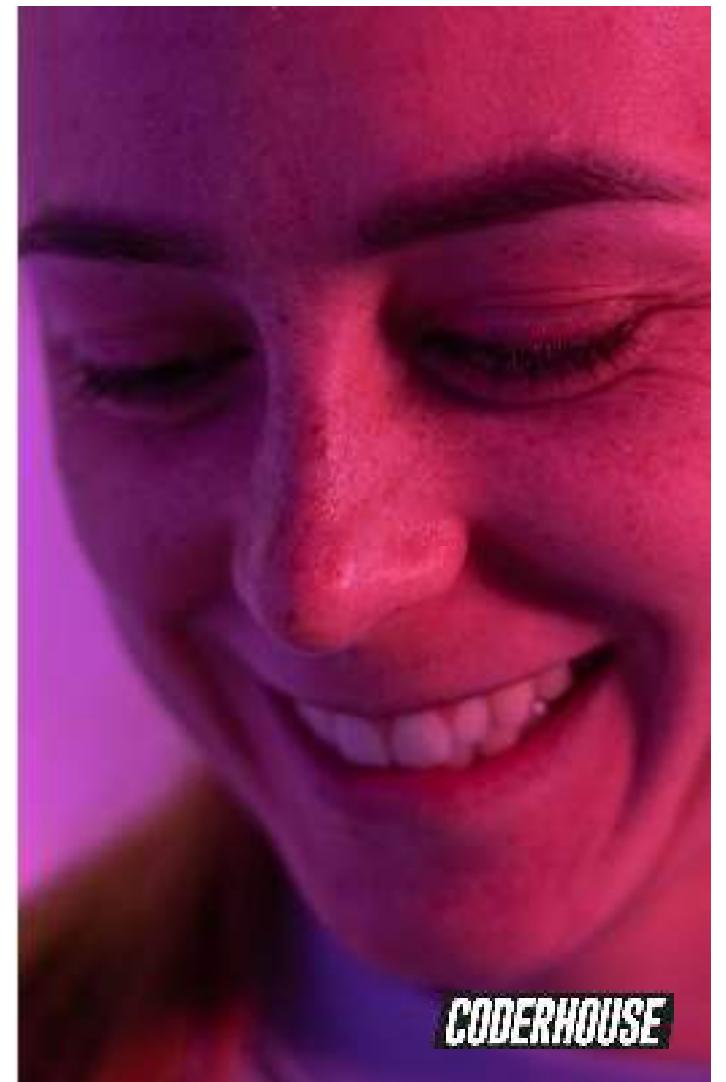
- | | |
|--------------|--------------|
| ○ Juan Pérez | ○ Juan Pérez |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |
| ○ ... | ○ ... |

CODERHOUSE

Presentación de estudiantes

Por encuestas de Zoom

1. País
2. Conocimientos previos
3. ¿Por qué elegiste este curso?



¿Dudas sobre el onboarding?

Míralo aquí

CODERHOUSE



Lo que debes saber antes de empezar

CODERHOUSE

¿Comenzamos?



CODERHOUSE

Acuerdos y compromisos

CODERHOUSE

ACUERDOS Y COMPROMISOS

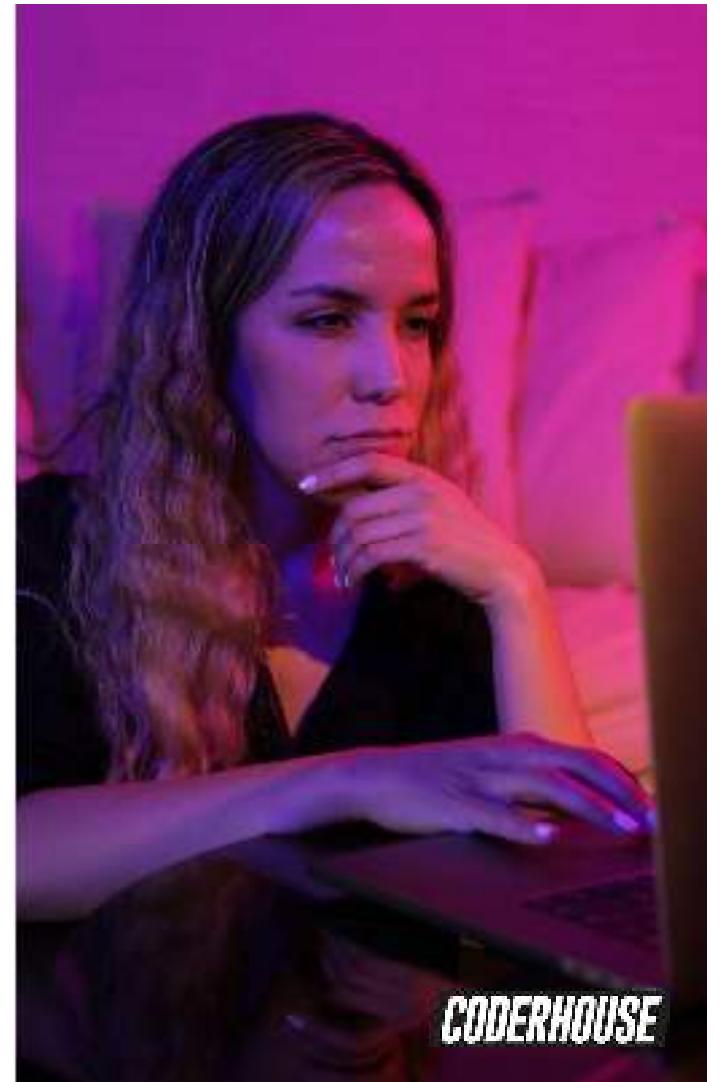
Convivencia

- ✓ Conoce aquí nuestro [código de conducta](#) y ayúdanos a generar un ambiente de clases súper ameno.
- ✓ Durante las clases, emplea los medios de comunicación oficiales para canalizar tus dudas, consultas y/o comentarios: **chat Zoom público y privado, y por el chat de la plataforma.**
- ✓ Ten en cuenta [las normas del buen hablante y del buen oyente](#), que nunca están de más.
- ✓ Verifica el estado de **la cámara y/o el micrófono** (on/off) de manera que esto no afecte la dinámica de la clase.

ACUERDOS Y COMPROMISOS

Distractores

- ✓ Encuentra tu espacio y crea el momento oportuno para **disfrutar de aprender**
- ✓ Evita dispositivos y aplicaciones que puedan **robar tu atención**
- ✓ Mantén la **mente abierta y flexible**, los prejuicios y paradigmas no están invitados

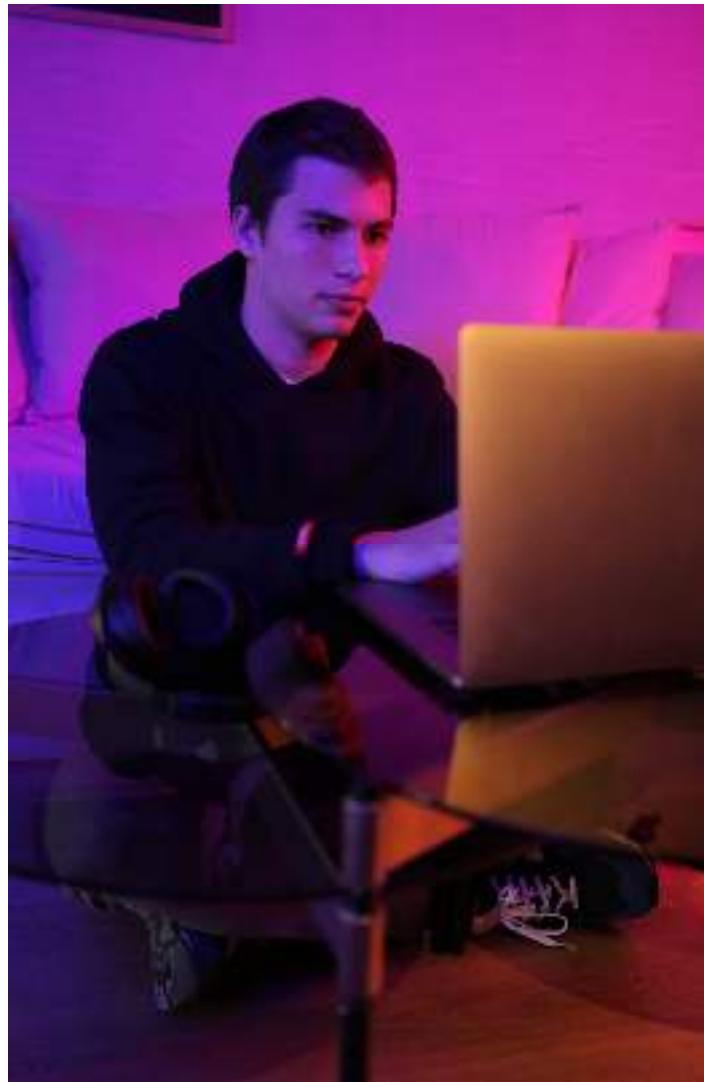


CODERHOUSE

ACUERDOS Y COMPROMISOS

Herramientas

- ✓ Mantén a tu alcance **agua, mate o café**
- ✓ Si lo necesitas, ten a mano lápiz y papel para que no se escapen las ideas. Pero recuerda que **en Google Drive tienes archivos que te ayudarán a repasar, incluidas las presentaciones.**
- ✓ Conéctate desde algún equipo (laptop, tablet) que te permita **realizar las actividades** sin complicaciones.
- ✓ Todas las clases quedarán grabadas y serán compartidas tanto en la **plataforma de Coderhouse como por Google Drive.**



ACUERDOS Y COMPROMISOS

Equipo

- ✓ ¡Participa de los After Class! Son un gran espacio para atender dudas y mostrar avances.
- ✓ Intercambia ideas por el **chat de la plataforma**.
- ✓ Siempre **interactúa respetuosamente**.
- ✓ No te olvides de **valorar tu experiencia** educativa y de contarnos cómo te va.

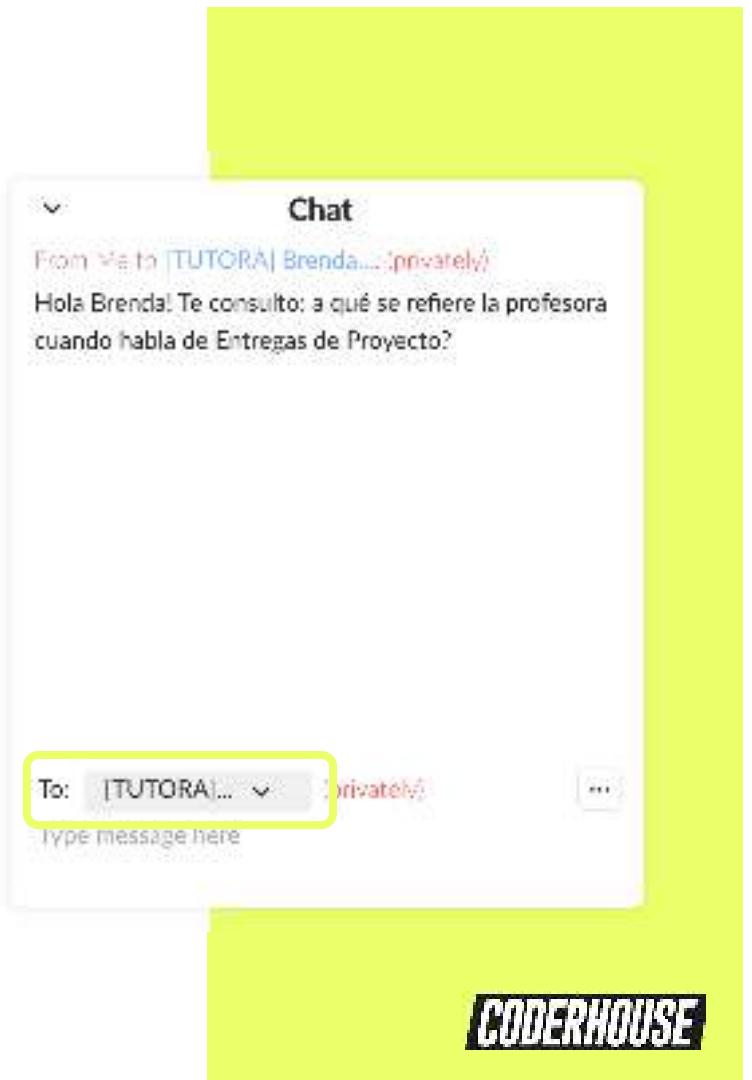
CODERHOUSE

INTERACCIONES EN CLASE

Mientras el profesor explica

Para mantener una comunicación clara y fluida a lo largo de la clase, te proponemos mantener 2 reglas:

1. Si tienes dudas durante la explicación, debes consultarle directamente por privado a tu tutor por el chat de Zoom.



CODERHOUSE

INTERACCIONES EN CLASE

Espacios para consultas

2. Entre contenido y contenido, se abrirán breves espacios de consulta. Allí puedes escribir en el chat tu pregunta.

¡Tu duda puede ayudar a otras personas!

No olvides seleccionar “todos” para que todos puedan leerte (y no solo tu tutor).



INTERACCIONES EN CLASE

Funcionalidades

Para **evitar saturar el chat de mensajes**, utiliza los signos que figuran en el apartado **Participantes**, dentro de Zoom.**



Por ejemplo: si se pregunta si se escucha correctamente, debes seleccionar la opción "Sí" o "No".



**Para quitar el signo, presiona el mismo botón nuevamente o la opción "clear all".

Clase 1. Testing y Escalabilidad Backend

Test y Mocks

CODERHOUSE

Temario

0

Proceso principal del servidor + Global & Child process

- ✓ Process
- ✓ Manejo de variables de entorno
- ✓ Global & child process

1

Test y Mocks

- ✓ TDD
- ✓ Desarrollo de prácticas preventivas a partir de mocks

2

Optimización

- ✓ Rendimiento en producción
- ✓ Comprensión con Brotli
- ✓ Middleware para manejo de errores

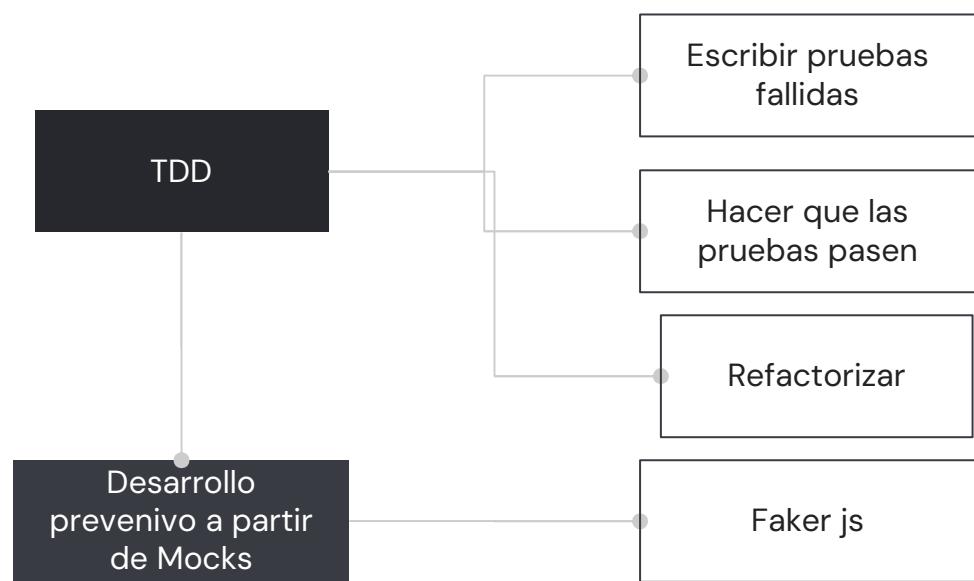
CODERHOUSE

Objetivos de la clase

- **Entender y aplicar** el concepto de TDD
- **Comprender** el concepto de Mocks
- **Realizar** un desarrollo práctico de Mocking.



MAPA DE CONCEPTOS



TDD

CODERHOUSE

Sobre el desarrollo y los errores

¿Cuántos errores has cometido en tu código desde que comenzaste a desarrollar? ¿Qué tan graves suelen ser esos errores?

Seguramente tus respuestas serán “**muchos**” y “**no muy graves**”. El nivel de gravedad de un error es bastante relativo, sin embargo, muchas veces lo podemos medir por qué tanto impacto tiene directamente en la experiencia del cliente.

Si aún no te has desenvuelto en un ambiente productivo, sabrás que todos los errores que se cometen no salen de tu computadora, es decir, todo se queda en un “entorno de desarrollo”.

CODERHOUSE

Errores en entorno productivo

¿Qué pasa cuando ocurre un error estando en un entorno productivo? Recordemos que este entorno es el nivel en el que el cliente puede visualizar e interactuar con todo, de manera que un error en este punto puede ser bastante grave.

Seamos desarrolladores frontend o backend, recordamos que la experiencia del usuario se puede ver afectada de manera que puede generar molestias al utilizar nuestra página, o incluso llevarlo a no querer volver a utilizarlo.



Tres principales errores:

- ✓ **De compilación:** "Mi código no compila", ocurre cuando un código no puede iniciarse debido a que el compilador encontró algún error antes de ejecutarlo.
- ✓ **De ejecución:** "Mi código explotó", ocurre cuando el código sí logra compilarse, pero a lo largo de su trabajo ocurre un error, usualmente no controlado.
- ✓ **Lógicos:** "Funciona, pero no funciona", ocurre cuando el código compila y se ejecuta correctamente, pero el resultado no es el esperado.

Consecuencias

- ✓ Las consecuencias podrán variar, algunas pueden ser muy simples...

"He estado experimentando que mi cámara tiembla descontroladamente cada vez que abro Snapchat o uso la cámara para Instagram", escribió uno. "Sin embargo, no tengo ningún problema cuando uso la aplicación de cámara normal".

- ✓ Y otros que ya no causan tanta gracia

Con base en los ingresos por US\$29.000 millones del último trimestre, *Fortune* señala que la **compañía perdió US\$99,75 millones en ingresos** por el número de horas inactivas de publicidad.

Fuente: [primera noticia](#) y [segunda noticia](#)



¡Piensa en tus pruebas!

Cuando nuestro código comienza a representar una empresa, organización, o cliente, cuya funcionalidad o errores definen las ganancias de la misma, tenemos que pensar que nuestro código, **no puede fallar**.

Pensar en un aplicativo con margen de error de 0% es utópico. Sin embargo, hacer pruebas de nuestro código reducirá en gran medida el margen de error de dicho módulo.



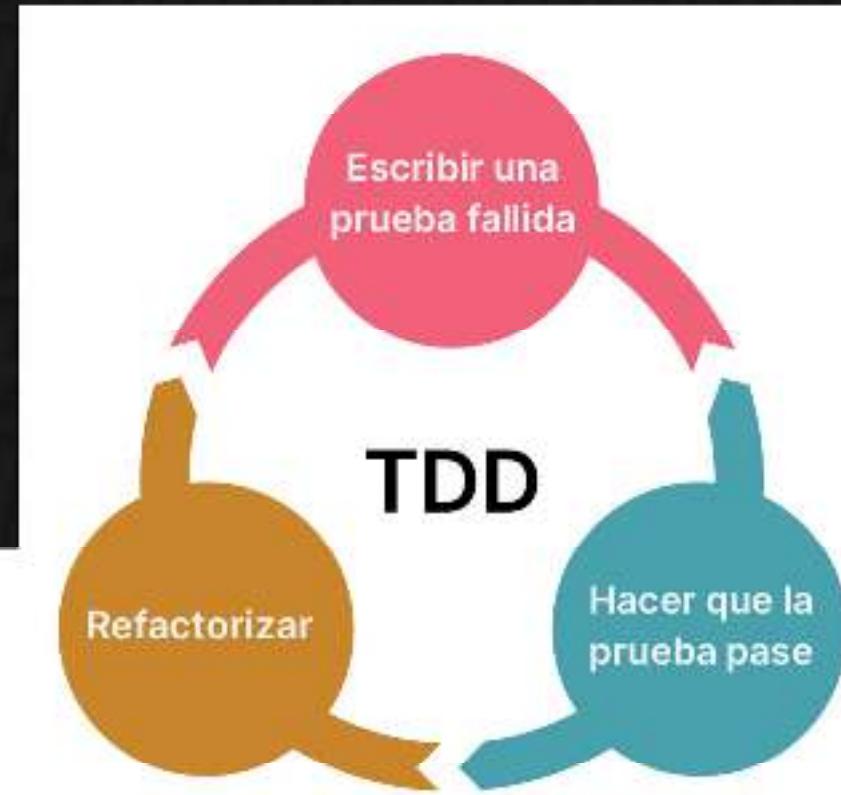
CODERHOUSE

Test Driven Development

Una de las mejores formas de abordar un problema, por muy grande o pequeño que sea, es a partir de una práctica de programación llamada **Test Driven Development (TDD, o desarrollo orientado a pruebas)**.

Este consiste en escribir las pruebas del módulo antes incluso de desarrollar dicho módulo, con el fin de dirigir nuestro desarrollo hacia el cumplimiento de estas pruebas. De esta manera, podemos visualizar cómo nuestro código irá cumpliendo punto a punto las cosas que debe abarcar en su uso.

El TDD es algo subjetivo y hay quienes agregan pasos extras y conceptos adicionales, pero en términos generales consiste en lo siguiente:



Etapa 1: Escribir pruebas fallidas

CODERHOUSE

Escribir una prueba fallida

Con alguna herramienta de testing, podemos colocar las pruebas que necesitemos con el fin de corroborar una funcionalidad.

Se dice que la primera prueba debe fallar, pues ya que nuestro código aún no ha sido implementado, no hay forma de que se pase. Por ejemplo, pensemos en una función:

```
JS suma.js  X
JS suma.js > [e] suma
1 const suma = () => [
2
3 ]
```

¿Qué pruebas hacer con una suma?

En este punto, podríamos comenzar a pensar en múltiples escenarios para poner a prueba.

Estos escenarios pueden hacerse desde un módulo de testing, sin embargo, por fines prácticos, lo haremos sin módulo y ejecutando directamente la función (No te preocupes, más adelante vamos a implementar módulos de testing reales).

Algunos escenarios a plantear podrían ser:

- ✓ La función debe devolver null si algún parámetro no es numérico.
- ✓ La función debe devolver 0 si no se pasó ningún parámetro
- ✓ La función debe poder realizar la suma correctamente.
- ✓ La función debe poder hacer la suma con cualquier cantidad de números.

Implementando el primer escenario

El primer punto a testear se podría hacer de la siguiente manera:

```
//1. La función debe devolver null si algún parámetro no es numérico
console.log("Test 1: La función debe devolver null si algún parámetro no es numérico")
let resultTest1 = suma("2",2);
if(resultTest1==null) {
  console.log("Test 1 pasado");
  testsPasados++;
}
```

Y, si corremos el código, podremos notar que no pasamos la primera prueba

```
Test 1 no pasado, se recibió undefined, pero se esperaba null
```

Implementando el resto de escenarios (1)

```
js suma.js > b6|resultest1
1 const suma = (num1,num2) =>{
2   if(!num1||!num2) return 0;
3   if(typeof num1!="number"||typeof num2!="number") return null;
4 }
5
6 //escenarios
7 let testsPasados = 0;
8 let testsTotales = 4;
9 //1. La función debe devolver null si algún parámetro no es numérico
10 console.log("Test 1: La función debe devolver null si algún parámetro no es numérico")
11 let resultTest1 = suma("1",2);
12 if(resultTest1==null){
13   console.log("Test 1 pasado");
14   testsPasados++;
15 }
16 else console.log("Test 1 no pasado, se recibió ${typeof resultTest1}, pero se esperaba null")
17 //2. La función debe devolver 0 si no se pasó ningún parámetro
18 console.log("Test 2: La función debe devolver 0 si no se pasó ningún parámetro")
19 let resultTest2 = suma();
20 if(resultTest2==0) {
21   console.log("Test 2 pasado");
22   testsPasados++;
23 }
24 else console.log("Test 2 no pasado, se recibió ${resultTest2}, pero se esperaba 0");
```

Implementando el resto de escenarios (2)

```
//3. La función debe poder realizar la suma correctamente
console.log("Test 3: La función debe resolver la suma correctamente.");
let resultTest3 = suma(2,3);
if(resultTest3==5) {
    console.log("Test 3 pasado");
    testsPasados++;
}
else console.log(`Test 3 no pasado, se recibió ${resultTest3}, pero se esperaba 6`);
//4. La función debe poder hacer la suma con cualquier cantidad de números
let resultTest4 = suma(1,2,3,4,5);
if(resultTest4==15) {
    console.log("Test 4 pasado");
    testsPasados++;
}
else console.log(`Test 4 no pasado, se recibió ${resultTest4}, pero se esperaba 15`);
if(testsPasados==testsTotales) console.log(`Todos los tests se han pasado con éxito`);
else console.log(`Se pasaron ${testsPasados} tests de un total de ${testsTotales}`);
```

Ejecutando escenarios

```
Test 1: La función debe devolver null si algún parámetro no es numérico
Test 1 no pasado, se recibió undefined, pero se esperaba null
Test 2: La función debe devolver 0 si no se pasó ningún parámetro
Test 2 no pasado, se recibió undefined, pero se esperaba 0
Test 3: La función debe resolver la suma correctamente.
Test 3 no pasado, se recibió undefined, pero se esperaba 6
Test 4 no pasado, se recibió undefined, pero se esperaba 15
Se pasaron 0 tests de un total de 4
```

Etapa 2: Hacer que las pruebas pasen

CODERHOUSE

Test 1: parámetro no numérico

Resolvamos por pasos, suponiéndose que se reciben dos valores numéricos, entonces podemos hacer:

```
const suma = (num1,num2) =>{
  if(typeof num1!=="number" || typeof num2!=="number") return null;
}
```

Hasta ahora entonces tenemos:

```
Test 1: La función debe devolver null si algún parámetro no es numérico
Test 1 pasado
Test 2: La función debe devolver 0 si no se pasó ningún parámetro
Test 2 no pasado, se recibió null, pero se esperaba 0
Test 3: La función debe resolver la suma correctamente.
Test 3 no pasado, se recibió undefined, pero se esperaba 6
Test 4 no pasado, se recibió undefined, pero se esperaba 15
Se pasaron 1 tests de un total de 4
```

Test 2: ningún parámetro

Podemos agregar una nueva validación para parámetros vacíos:

```
const suma = (num1,num2) =>{
  if(!num1||!num2) return 0;
  if(typeof num1!=="number"||typeof num2!=="number") return null;
}
```

Hasta ahora entonces tenemos:

```
Test 1: La función debe devolver null si algún parámetro no es numérico
Test 1 pasado
Test 2: La función debe devolver 0 si no se pasó ningún parámetro
Test 2 pasado
```

Test 3: Suma

Ejecutamos el código necesario para hacer la suma (no importa la optimización)

```
const suma = (num1,num2) =>{
    if(!num1||!num2) return 0;
    if(typeof num1!="number"||typeof num2!="number") return null;
    let result = num1+num2;
    return result;
}
```

Ya casi finalizamos:

```
Test 1: La función debe devolver null si algún parámetro no es numérico
Test 1 pasado
Test 2: La función debe devolver 0 si no se pasó ningún parámetro
Test 2 pasado
Test 3: La función debe resolver la suma correctamente.
Test 3 pasado
Test 4 no pasado, se recibió 3, pero se esperaba 15
Se pasaron 3 tests de un total de 4
```

Test 4: ¿ n parámetros?

Notamos que en este punto será necesario hacer cambios más grandes, ya que para n número de parámetros, no servirán las validaciones que tenemos.

Entonces toca reformular el problema. ¿Cómo trabajar con cualquier número de inputs? Procedemos a los cambios:

```
1 const suma = (...nums) =>{
2     if(nums.length === 0) return 0;
3     let validInput = true;
4     for(let i = 0; i < nums.length && validInput; i++){
5         if(typeof nums[i] !== "number"){
6             validInput=false;
7         }
8     }
9     if(!validInput) return null;
10    let result = 0;
11    for(let i=0;i<nums.length;i++){
12        result+=nums[i];
13    }
14    return result;
15 }
```

Etapa 3: Refactorizar

CODERHOUSE

¡Listo!

¡Logramos pasar nuestros tests! Sin embargo, creo que notamos algo bastante serio:

El código funciona, de manera que pasamos las pruebas necesarias para considerarse “funcional”, sin embargo, ¿es óptimo?

Tenemos que buscar ahora una forma de que nuestro código sea más limpio y sintetizar el código.

```
Test 1: La función debe devolver null si algún parámetro no es numérico
Test 1 pasado
Test 2: La función debe devolver 0 si no se pasó ningún parámetro
Test 2 pasado
Test 3: La función debe resolver la suma correctamente.
Test 3 pasado
Test 4: La función debe poder funcionar con cualquier cantidad de números
Test 4 pasado
Todos los tests se han pasado con éxito
```

Sobre el arte de la refactorización

Al refactorizar, no pensamos en nuevas funcionalidades ni nada extra al código que tenemos, sino que **buscamos la manera de poder sintetizar tareas que podrían implementarse de otra forma.**

Al refactorizar, conseguimos mantener una funcionalidad idéntica para no afectar los tests, pero realizando tareas mucho más limpias.

Es la etapa final del TDD, una vez realizado ésto, podemos dar nuestro código por finalizado. 🎉

Refactorización

Función suma antes

```
1 const suma = (...nums) =>{
2   if(nums.length === 0) return 0;
3   let validInput = true;
4   for(let i = 0; i < nums.length && validInput; i++){
5     if(typeof nums[i] !== "number"){
6       validInput = false;
7     }
8   }
9   if(!validInput) return null;
10  let result = 0;
11  for(let i=0;i<nums.length;i++){
12    result+=nums[i];
13  }
14  return result;
15 }
```

Función suma ahora

```
const suma = (...nums) =>{
  if(nums.length === 0) return 0;
  if(!nums.every(num=>typeof num === "number")) return null;
  return nums.reduce((prev,current)=>prev+current)
}
```

No hay afeccción en los tests

Todos los tests se han pasado con éxito

Listo, ¡Un módulo desarrollado a partir de un estilo de trabajo TDD!

El desarrollo basado en tests es bastante solicitado en nivel empresarial, ya que es una práctica que reduce en gran medida la posibilidad de errores “no contemplados”.

Evidentemente no es posible escribir TODOS los tests para TODOS los escenarios, pero es de gran apoyo.

Existen múltiples formas de testear estas funcionalidades, las cuales abordaremos más adelante en el módulo, para darte la seguridad de testear tu código y entregar un trabajo de calidad en tu trabajo.

¡Importante!

El TDD no es una herramienta, no es un módulo externo, no es algo de internet que puede implementarse. Es una forma de programar y, por lo tanto, necesitarás profundizar, practicarlo e implementarlo en entornos reales por tu cuenta.



Aplicando TDD

Duración: 20–25 min

CODERHOUSE



ACTIVIDAD EN CLASE

Aplicando TDD

Aplicar bajo el modelo de trabajo de TDD:

Una función de login (con usuarios hardcodeados user = coderUser , password = 123)

- ✓ Si se pasa un password vacío, la función debe consologuear ("No se ha proporcionado un password")
- ✓ Si se pasa un usuario vacío, la función debe consologuear ("No se ha proporcionado un usuario")
- ✓ Si se pasa un password incorrecto, consologuear ("Contraseña incorrecta")
- ✓ Si se pasa un usuario incorrecto, consologuear ("Credenciales incorrectas")
- ✓ Si el usuario y contraseña coinciden, consologuear ("logueado")



Break

¡10 minutos y volvemos!

CODERHOUSE

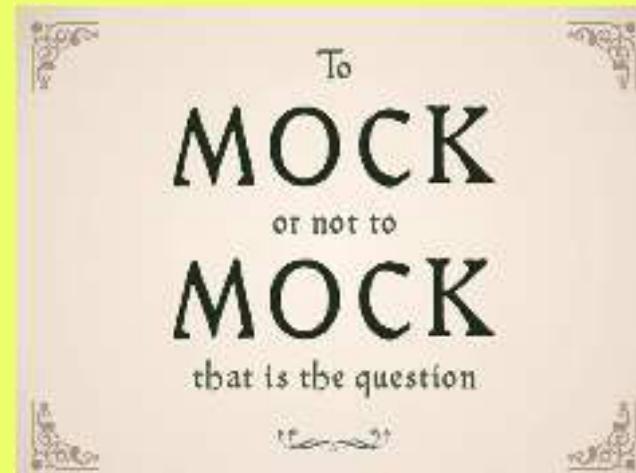
Desarrollo preventivo a partir de mocks

CODERHOUSE

¿Qué es un mock?

Dentro de sus múltiples traducciones, podemos tomar **mock** como una “imitación” de un dato real. Nos es altamente útil para poder crear datos “supuestos” con el fin de probar la funcionalidad de alguna función.

Un dato mock no debe comprometer jamás una estructura productiva, por lo que solo se usa en entornos de desarrollo.



Usos comunes de mocks

El escenario bajo el cual se suelen usar mocks son:

- ✓ Para generar usuarios falsos y probar módulos de registros o logins y sesiones.
- ✓ Para generar productos de prueba en un ecommerce.
- ✓ NPCs o personajes prueba para algún videojuego.
- ✓ Conjuntos de estadísticas para módulos de análisis
- ✓ Conjuntos de coordenadas para sistemas de geolocalización

Faker - js

CODERHOUSE

Faker-js

Faker-js es un módulo externo de node js derivado del original **fakerjs**, el cual es un módulo pensado para poder realizar modelos de prueba de diferentes formas, en diferentes idiomas y diferentes escenarios.

¡Faker es extremadamente potente!

Podemos crear mocks de cosas como:

- ✓ nombres, apellidos, edades
- ✓ colores, fechas, números
- ✓ animales, países, géneros musicales
- ✓ vehículos,Ipv4s, coordenadas geográficas
- ✓ ¡Es un mundo de datos para probar!



Algunas de las cosas que podemos simular con faker

Address	Date	Lorem
buildingNumber	between	line
cardinalDirection	between	paragraph
city	birthdate	paragraphs
cityName	future	sentence
cityPrefix	month	sentences
citySuffix	past	slug
country	recent	text
countryCode	soon	word
county	weekday	words
direction		
latitude		
longitude		
nearbyGPSCoordinate		
ordinalDirection	Faker	Mersenne
secondaryAddress	fake	rand
state		seed
stateAbb		seedArray
street		
streetAddress		
streetName	Finance	
streetPref	account	
streetSuffix	accountName	
timeZone	amount	
zipCode	btc	
zipCodeByState	bitcoinAddress	
	creditCardCVV	Music
	creditCardIssuer	genre
	creditCardNumber	songName
		Name

Referencia completa de la página [aquí](#)

CODERHOUSE

Caso práctico

En la startup que trabajamos, el equipo de data science y analytics necesita realizar algunos algoritmos de análisis a partir de los usuarios que tenemos en nuestra base de datos.

Sin embargo, ya que la empresa es nueva, no contamos con una base lo suficientemente amplia para poder alimentar las analíticas del algoritmo que están desarrollando.

Necesitaremos “simular” múltiples usuarios con el fin de que la respuesta tenga una cantidad de usuarios lo suficientemente abundante para que ellos testeen sus propios algoritmos





Hands on lab

En esta instancia de la clase **repasaremos** algunos de los conceptos vistos en clase con una aplicación

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **30 minutos**

CODERHOUSE



HANDS ON LAB

Mocking API

¿Cómo lo hacemos? **Desarrollaremos un servidor que pueda devolver los datos de prueba que necesitamos para poder presentarlos al equipo correspondiente, para ésto, tendremos que considerar los siguientes elementos:**

- ✓ Necesitaremos crear un modelo de productos de prueba para alimentar los usuarios de prueba.
- ✓ Además, necesitaremos crear usuarios de prueba, cuyo carrito corresponda a un arreglo alimentado por los productos mock creados previamente.
- ✓ Contaremos con un endpoint /api/users, el cual se encargará de devolver a los usuarios de prueba.
- ✓ Además, tendremos una función “generateUsers” y una función “generateProducts”

Snapshot: app.js

```
JS app.js    ✘  
src > JS app.js > ...  
1 import express from 'express';  
2 import usersRouter from './routes/users.js'  
3 const app = express();  
4 const PORT = process.env.PORT || 8080;  
5  
6 app.use('/api/users',usersRouter);  
7 app.listen(PORT,()=>console.log(`Listening on ${PORT}`))
```

Snapshot: users.js

```
JS app.js          JS users.js  X
src > routes > JS users.js > [o] default
1  import { Router } from 'express';
2  import { generateUser } from '../utils.js';
3
4  const router = Router();
5
6  router.get('/',async(req,res)=>{
7      let users = []
8      for(let i=0;i<100;i++){
9          users.push(generateUser())
10     }
11     res.send({status:"success",payload:users})
12   })
13
14  export default router;
```

Snapshot: utils.js (generateUser)

Recuerda que aquí comienza la magia del mocking, así que hay que hacer npm install del módulo de faker-js, el cual se hace:

```
npm install @faker-js/faker
```

```
1 import { Faker } from '@faker-js/faker';
2
3 export const generateUser = () => {
4   let numOfProducts = faker.number.int({min: 1, max: 7});
5   let products = [];
6   for (let i = 0; i < numOfProducts; i++) {
7     products.push(generateProduct());
8   }
9
10  return {
11    id: faker.database.mongodbObjectId(),
12    name: faker.person.firstName(),
13    last_name: faker.person.lastName(),
14    sex: faker.person.sex(),
15    birthDate: faker.date.birthdate(),
16    phone: faker.phone.number(),
17    products,
18    image: faker.internet.avatar(),
19    email: faker.internet.email()
20  };
21}
```

Snapshot: utils.js (generateProduct)

```
23  export const generateProduct = () => {
24    return {
25      id: faker.database.mongodbObjectId(),
26      title: faker.commerce.productName(),
27      price: faker.commerce.price(),
28      department: faker.commerce.department(),
29      stock: faker.number.int({min: 0, max: 100}),
30      image: faker.image.url()
31    }
32  }
```

Snapshot: respuesta del endpoint (¡Cuántos datos!)



Más peticiones

Duración: 20–25 min

CODERHOUSE



ACTIVIDAD EN CLASE

Más peticiones

A partir del servidor recién desarrollado

Se nos han solicitado algunas modificaciones para con el desarrollo del mocking:

- ✓ Ahora la generación de usuarios necesita que se separen por roles, los posibles roles son:
 - cliente
 - vendedor
- ✓ Un booleano que indique si el usuario es premium (no importando el rol)
- ✓ El producto debe tener un campo “code” que sea alfanumérico
- ✓ El producto debe contar con una breve descripción, ya sea por lorem o por producto.
- ✓ El usuario debe mostrar su actual ocupación laboral.

¿Cómo sé dónde están estos campos? Vamos a investigar <https://fakerjs.dev/api/>

¡Caso resuelto!

¡Un día de trabajo más! ahora podemos generar 50-100-1000-10000 datos diferentes y todo lo necesario para que otros sectores puedan seguir trabajando.

En este caso particular estamos haciéndolo en un proyecto aislado, ¡más adelante tocará aplicarlo para funcionalidades más complejas, como pruebas de carga, de funcionalidades, etc.



¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Qué es el TDD
- ✓ Qué son los mocks
- ✓ faker js
- ✓ Aplicar los conceptos en una mocking API sencilla.

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser

- grabada

Clase 10. Escalabilidad y Testing Backend

Testing avanzado

CODERHOUSE

Temario

09

Testing Unitario

- ✓ Módulos de testing
- ✓ Testing con Mocha
- ✓ Testing con Chai

10

Testing Avanzado

- ✓ Tests de integración
- ✓ Tests con supertest
- ✓ Testing de elementos avanzados

11

Frameworks de desarrollo Parte I

- ✓ Framework de desarrollo
- ✓ NESTjs
- ✓ Métodos principales en NESTjs

Objetivos de la clase

- Diferencia entre tests unitarios y tests de integración
- Comprender la librería SuperTest
- Desarrollo de un flujo de testing con Mocha + Chai + SuperTest

CLASE N°09

Glosario

TERMINOLOGÍA ELEMENTAL DE TESTING

Assert: módulo nativo de nodejs que nos permitirá hacer validaciones de manera estricta.

archivo.test.js: la subextensión .test.js indica que el archivo será utilizado dentro de un contexto de testing

describe: función utilizada para definir diferentes contextos de testeo, podemos tener la cantidad de contextos que deseemos en un flujo de testing, siempre y cuando reflejen intenciones diferentes.

it: unidad mínima de nuestro testing, en ella, definimos qué acción se está realizando y cuál será el resultado esperado.

Mocha: Es un framework de testing originalmente diseñado para nodejs, el cual nos permitirá ejecutar entornos completos para poder hacer cualquier tipo de pruebas que necesitemos.

Test unitario: está pensado para funcionalidades aisladas, es decir, aquellas funcionalidades en las que **no se consideran el contexto u otros componentes**.

CODERHOUSE

CLASE N°09

Glosario

TERMINOLOGÍA ELEMENTAL DE TESTING

before: Función que nos permite inicializar elementos antes de comenzar con todo el contexto de testeo.

beforeEach: Función que nos permite inicializar elementos antes de comenzar **cada test** dentro de un contexto particular.

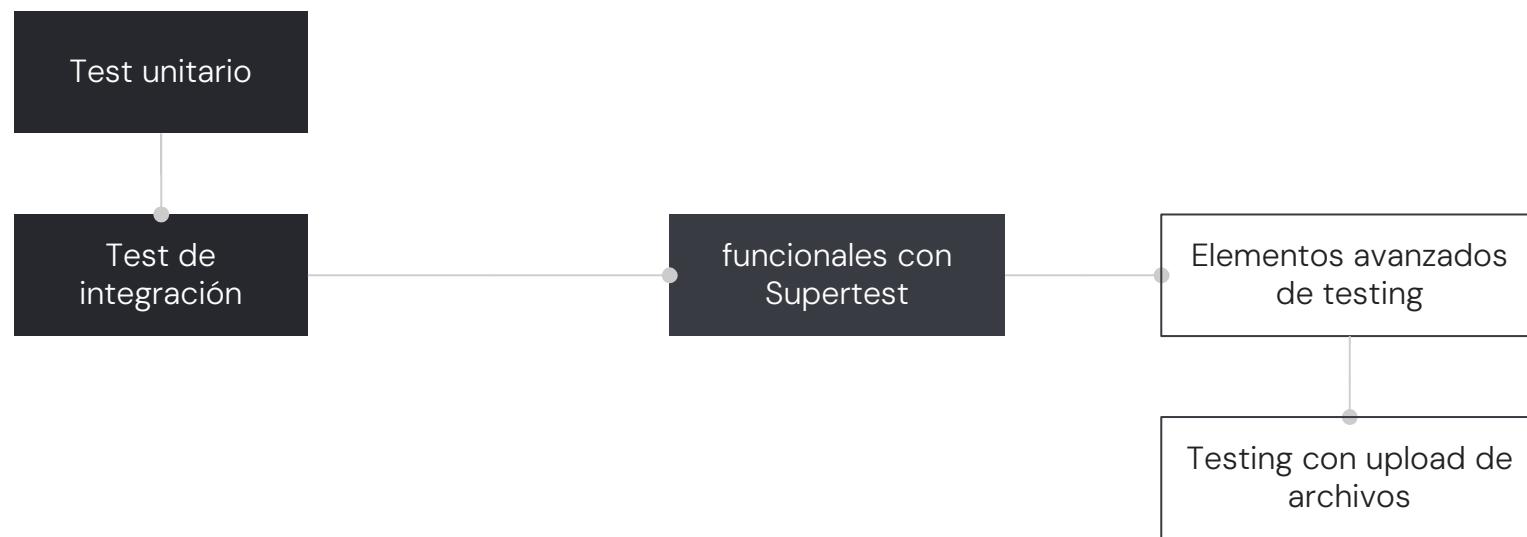
after: Función que nos permite realizar alguna acción una vez finalizado el contexto de testeo

afterEach: Función que nos permite realizar alguna acción una vez finalizado **cada test** dentro del contexto particular.

CODERHOUSE



MAPA DE CONCEPTOS



Test unitario vs Test de integración

CODERHOUSE

Sobre el test unitario

Cuando hablamos de test unitario, nos referimos al elemento más pequeño que puede ser testeado. La intención de las pruebas va dirigida al correcto funcionamiento de un módulo isolado.

La principal característica de trabajar una prueba unitaria es corroborar que los detalles más pequeños de dicho módulo sean cubiertos, sin embargo, el correcto funcionamiento de éste no contempla otros módulos dentro del aplicativo.

Hacer tests unitarios es útil para no tener que regresar a revisar los detalles más pequeños cuando estamos trabajando entornos más robustos.

Solos funcionan, ¿qué tal en conjunto?

Analiza la imagen que se te presenta a la derecha. En este caso tenemos dos módulos: El seguro de una puerta y la puerta en sí.

El seguro cumple su función al poder hacerse a un lado y poder llegar a un extremo. La puerta cumple su función al poder abrirse y cerrarse. Sin embargo, cuando ambos están trabajando en conjunto, el comportamiento ya no es satisfactorio.

Podemos decir que los elementos, a pesar de funcionar correctamente de manera **unitaria**, no pasaron la prueba de **integración**.



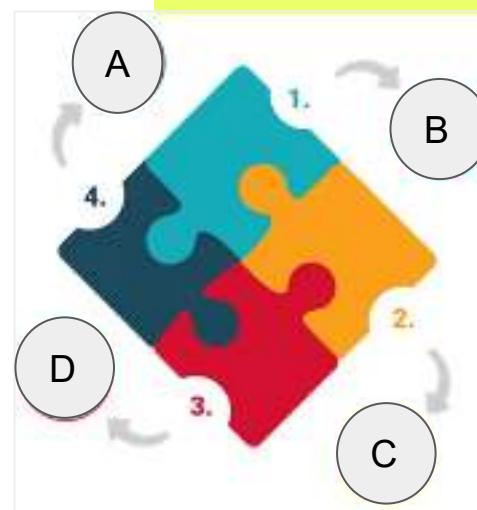
Sobre el testing de integración

Como ya imaginarás, un test de integración tiene el objetivo de ver que los módulos funcionen **en conjunto**. Así, las funcionalidades conjuntas llevan a un resultado más complejo, en menor o mayor medida.

Se traduce a cualquier tarea que podamos probar, donde los módulos puedan mezclar sus tareas individuales, para generar un trabajo en conjunto.

Por ejemplo:

- ✓ **Módulo de Dao**: Permite guardar correctamente un usuario en la base de datos.
- ✓ **Bcrypt**: permite hashear correctamente una string.
- ✓ **Test de integración**: El conjunto de los módulos permite que se guarde un usuario en la base de datos, con su contraseña hasheada.



Testing de carácter funcional

Existe un punto más allá del test de integración (considerado por algunos desarrolladores como algo externo, aunque algunos otros desarrolladores lo consideran dentro del mismo proceso de integración).

El tipo de test conocido como **test funcional**, hace referencia a aplicar las integraciones de la misma manera que se realizaría un test de integración, sin embargo, éstas enfocadas a cumplir una funcionalidad real.

Por ejemplo, si integramos el Dao de usuarios, con bcrypt, además de testear el módulo de routing, controlador de express y un middleware de passport, podríamos llegar a generar una integración lo suficientemente sólida para concretar una funcionalidad: el registro de un usuario o el login del mismo.

Nota como en el ejemplo enunciado hablamos de hacer una integración, sí, pero esta vez enfocada a un resultado más complejo en un proceso más robusto.

Algunos lo entienden como un **súper test de integración**.

CODERHOUSE

Testings de integración y funcionales con Supertest

CODERHOUSE

Supertest

En la última diapositiva mencionamos que podíamos integrar las pruebas de módulos con routers y controladores. ¿Pero cómo probamos de manera directa la funcionalidad de nuestro servidor?

Supertest es una librería que nos permitirá ejecutar peticiones HTTP a nuestro servidor, para poder probar funcionalidades como estatus de peticiones, envío de bodies en petición o revisión de respuestas recibidas por el servidor.

Al probar un endpoint, estaremos probando múltiples módulos en conjunto, utilizados para resolver la funcionalidad que refleja el endpoint.



CODERHOUSE

Inicializando un test con supertest

Lo primero, como ya será costumbre, es utilizar el proyecto de Adoptme que hemos utilizado en clases previas. Puedes utilizar el proyecto que teníamos de la clase pasada, en ese caso, sólo será necesario instalar supertest:

```
npm install -D supertest
```

Sin embargo, si deseas clonar nuevamente el proyecto de raíz, la instalación de supertest vendrá acompañada con mocha y chai.

```
npm install -D mocha chai supertest
```



Checkpoint: Espacio de preparación

Toma un tiempo para clonar o inicializar el proyecto [Aquí](#).
Revisa el código e instala las dependencias indicadas.

Tiempo estimado: 3 minutos

CODERHOUSE

Preparamos una doble terminal

Vamos a probar nuestros endpoints, por lo que se recomienda que tengas dos terminales.



```
[nodemon] starting "node src/app.js"
[nodemon] restarting due to changes...
[nodemon] starting "node src/app.js"
[nodemon] restarting due to changes...
[nodemon] starting "node src/app.js"
[nodemon] restarting due to changes...
[nodemon] starting "node src/app.js"
[nodemon] restarting due to changes...
[nodemon] starting "node src/app.js"
[nodemon] restarting due to changes...
[nodemon] starting "node src/app.js"
[nodemon] listening on 8080
D:
```

- ✓ La primera estará pensada para ejecutar el servidor y dejarlo escuchando, listo para recibir las peticiones de nuestro test.
- ✓ La otra terminal servirá para ejecutar el comando de test las veces que sean necesarias hasta finalizar con el flujo de pruebas.

Obtengamos los elementos para nuestras pruebas

Ya que mandaremos a llamar los endpoints de nuestro servidor, no será necesario sacar el Dao como lo hicimos en las pruebas de la clase pasada, esta vez bastará con importar chai y supertest para comenzar con nuestras pruebas.

Recordemos que chai servirá para hacer las pruebas a partir de **expect**, el cual declaramos en la línea 4.

Además, supertest puede generar un **requester**. Este requester de la línea 5 será el encargado de realizar las peticiones al servidor.

```
JS supertesttest.js U X
test > JS supertesttest.js > ...
1 import chai from 'chai';
2 import supertest from 'supertest';
3
4 const expect = chai.expect;
5 const requester = supertest('http://localhost:8080')
```

Probando el módulo de mascotas (Pet)

Contamos con dos describes: el primero será referente a todo el entorno de Adoptme, sin embargo, en esta ocasión tendremos diferentes **describe** internos para cada módulo (un **describe** para mascotas, uno para usuarios, etc).

Primero probaremos creando una mascota llamando a POST /api/pets

```
describe('Testing Adoptme', ()=>{
  describe('Test de mascotas', ()=>{
    it('El endpoint POST /api/pets debe crear una mascota correctamente', async ()=>{
      // ...
    })
  })
})
```

Primer request

Escribimos el primer test para crear una mascota, algunas de las propiedades que podemos obtener de la respuesta del requester son:

- ✓ Statuscode: Código del status.
- ✓ Ok: Si el status corresponde a un valor ok (es igual para Redirected, unauthorized, forbidden, etc)
- ✓ _body: Permite obtener el cuerpo de la respuesta (podemos ver el status y el payload en la consola)

```
5 supertest.test.js ×
test > 5 supertest.test.js > ② describe('Testing Adoptme') callback > ③ describe('Test de mascotas') callback > ④ it('El endpoint POST /api/pets debe crear una
    / describe('Testing Adoptme', () => {
      ③ describe('Test de mascotas', () => {
        ④ it('El endpoint POST /api/pets debe crear una mascota correctamente', async() => {
          const petMock = {
            name: "Pattitas",
            specie: "Perro",
            birthDate: "18-10-2022",
          }
          const {
            statusCode,
            ok,
            _body
          } = await requester.post('/api/pets').send(petMock)
          console.log(statusCode);
          console.log(ok);
          console.log(_body);
        })
      })
    })
  })
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

> mocha test/supertest.test.js

```
Testing Adoptme
  Test de mascotas
200
true
{
  status: 'success',
  payload: {
    name: 'Pattitas',
    specie: 'Perro',
  }
}
```

Evaluando el test

Escribimos el primer test para crear una mascota, algunas de las propiedades que podemos obtener de la respuesta del requester son:

- ✓ Statuscode: Código del status.
- ✓ Ok: Si el status corresponde a un valor ok (es igual para Redirected, unauthorized, forbidden, etc)
- ✓ _body: Permite obtener el cuerpo de la respuesta (podemos ver el status y el payload en la consola)

```
statusCode,  
ok,  
_body  
} = await requester.post('/api/pets').send(petMock)  
//Preguntamos si el payload tiene un _id, en caso de tenerlo,  
// significa que la creación fue correcta  
expect(_body.payload).to.have.property('_id');
```

Resumiendo...

A partir del requester de supertest podemos probar cada uno de los endpoints que necesitemos.

Una vez obtenida la respuesta, el resto se trata de hacer las validaciones a partir de Chai.

La combinación de las librerías mencionadas nos permitirá definir un flujo suficientemente sólido para poder realizar todo entorno de pruebas que nosotros necesitemos.

A continuación, deberás poner en práctica la combinación de estas librerías para poder realizar el resto de pruebas del módulo de mascotas.



Pruebas del módulo Pets

Duración: 15/20 min



ACTIVIDAD EN CLASE

Pruebas del módulo Pets

Sobre el mismo archivo donde probaremos nuestro servidor.

Continuar con el flujo del módulo de mascotas (Pets) para poder realizar las siguientes pruebas.

- ✓ Al crear una mascota sólo con los datos elementales. Se debe corroborar que la mascota creada cuente con una propiedad **adopted : false**
- ✓ Si se desea crear una mascota sin el campo **nombre**, el módulo debe responder con un status 400.
- ✓ Al obtener a las mascotas con el método GET, la respuesta debe tener los campos status y payload. Además, payload debe ser de tipo arreglo.
- ✓ El método PUT debe poder actualizar correctamente a una mascota determinada (esto se puede testear comparando el valor previo con el nuevo valor de la base de datos).
- ✓ El método DELETE debe poder borrar la última mascota agregada, ésto se puede alcanzar agregando a la mascota con un POST, tomando el id, borrando la mascota con el DELETE, y luego corroborar si la mascota existe con un GET



Para pensar

¿Qué alternativa has utilizado para resolver la actividad?

¿Has podido comprobar que existen múltiples maneras de resolver un proceso de Testing?

Contesta en el chat de Zoom



Break

¡10 minutos y volvemos!

CODERHOUSE

Elementos avanzados de Testing

CODERHOUSE

No todos los endpoints son tan sencillos de resolver

Algunos de los endpoints que desarrollamos tienen una lógica de desarrollo más compleja de lo normal, por ejemplo, sabemos que el endpoint **/api/sessions/login** No cuenta sólo con una respuesta simple, sino que además éste setea una cookie para resolución de sesión con un token de jwt.

Además, existen endpoints que reciben más que sólo información (por ejemplo, el registro de un usuario con un avatar, deberá enviarse con un archivo).

Tu proyecto Adoptme tiene algunos de estos endpoints para probar, de manera que desarrollaremos la lógica de tests para estos casos particulares

Corroborar que el login devuelva una cookie

Lo primero es evaluar elementos más allá de la respuesta original.

Esta vez realizaremos un flujo de testing basado en el router de sessions.

- ✓ Primero realizaremos un registro.
- ✓ Posteriormente, con el mismo usuario registrado, llamaremos a nuestro login
- ✓ A partir del login, no evaluaremos necesariamente la respuesta, sino que también nuestro punto de interés será recibir una cookie con el usuario.
- ✓ Esta cookie la utilizaremos posteriormente para probar que el endpoint **current** reciba la cookie y nos entregue la información que necesitamos.



Test 1: registro del usuario

Además, declaramos una variable “cookie” de manera global en el contexto describe, la utilizaremos para el siguiente test.

```
describe('Test avanzado', () => {
  let cookie;
  it('Debe registrar correctamente a un usuario', async function () {
    const mockUser = {
      first_name: "Mauricio",
      last_name: "Espinosa",
      email: "correomau@correo.com",
      password: "123"
    }
    const { _body } = await requester.post('/api/sessions/register').send(mockUser);
    //Sólo nos basta que esté definido el payload, indicando que tiene un _id registrado
    expect(_body.payload).to.be.ok;
  })
})
```

Test 2: Login con los datos del usuario

Nos interesa esperar (expect) 3 cosas: Que el resultado de la cookie realmente funcione, que la cookie final tenga el nombre de "coderCookie" (que es el nombre que se setea desde el endpoint), y que el valor esté definido.

```
it('Debe loguear correctamente al usuario Y DEVOLVER UNA COOKIE', async function() {
    //Enviamos al login los mismos datos del usuario que recién registramos.
    const mockUser = {
        email: 'correo@mail.com',
        password: '123'
    }
    //Ahora, Obtenemos de supertest los headers de la respuesta y extraeremos el header "set-cookie"
    //En caso de que éste venga correctamente, significa que el endpoint efectivamente devuelve una cookie.
    //Guardaremos el valor de la cookie en la variable "cookie" declarada arriba.
    const result = await requester.post('/api/sessions/login').send(mockUser);
    const cookieResult = result.headers['set-cookie'][0]
    expect(cookieResult).to.be.ok;
    cookie = {
        name: cookieResult.split(';')[0],
        value: cookieResult.split(';')[1]
    }
    expect(cookie.name).to.be.ok.and.eql('coderCookie');
    expect(cookie.value).to.be.ok;
})
```

Test 3: Enviando la cookie recibida por el login

Es el último endpoint a probar para esta sesión, indicando que, cuando envíe la cookie al servidor, este debería traerme el usuario guardado en el token. Con esto cumplimos el flujo de register, login y current.

```
it('Debe enviar la cookie que contiene el usuario y destrucutar éste correctamente', async function () {
  //Enviamos la cookie que guardamos arriba a partir de un set.
  const {_body} = await requester.get('/api/sessions/current').set('Cookie', [` ${cookie.name}=${cookie.value}`])
  //Luego, el método current debería devolver el correo del usuario que se guardó desde el login.
  //Indicando que efectivamente se guardó una cookie con el valor del usuario (correo).
  expect(_body.payload.email).to.be.eql('correo@correo.com');
})
```

Importante

A pesar de que **existen otras formas de pasar una cookie de manera directa entre un test y otros**, estos métodos usualmente se encuentran generando una mini instancia de app en el mismo, cosa que **no es necesario realizar si nuestro servidor ya corre en una terminal paralela**.



Test de rutas desprotegidas

Duración: 10 min



ACTIVIDAD EN CLASE

Test de rutas desprotegidas

Con base en el proyecto Adoptme que actualmente estamos utilizando

Existen dos endpoints: `/unprotectedLogin` y `/unprotectedCurrent` en el router de sessions. Evaluar:

- ✓ Que el endpoint de `unprotectedLogin` devuelva una cookie de nombre **`unprotectedCookie`**.
- ✓ Que el endpoint `unprotectedCurrent` devuelva al usuario completo, evaluar que se encuentren todos los campos que se guardaron en la base de datos.



Testing con upload de archivos

CODERHOUSE

El upload de archivos es otro tema de cuidado

Como último punto para testear nuestras funcionalidades, está el upload de archivos.

Pueden ser probados a partir de uploads locales con multer en diskStorage, o bien complementados a partir de un sistema de almacenamiento externo, como sharepoint, google drive, o AWS S3.

En esta ocasión utilizaremos diskStorage.



Ya contamos con un endpoint

Para poder trabajar con el test de archivos, ocuparemos un endpoint que ya se encuentra en el proyecto de Adoptme:
/api/pets/withimage

Tiene todo lo necesario para trabajar con multer, el cual ya se encuentra instalado en el proyecto, basta con enviarle un test en tipo FormData para que sea leído correctamente en el endpoint.

Solo nos queda enseñar a SuperTest a enviar una imagen para cargar. ↗



CODERHOUSE

Recordemos que no enviamos un json

QUESTION

Cuando estamos trabajando con archivos, recuerda que no es factible poder enviar archivos por medio de un json. Es por ello que, para este caso, tendremos que enviar un ***multipart FormData***.

Nos olvidaremos un poco del .send que hemos utilizado con nuestro requester, para cambiar la lógica de envío.

Preparemos el test con nuestro objeto mascota de prueba.

```
/*  
describe('Test uploads', ()=>{  
    it("Debe poder crearse una mascota con la ruta de la imagen", async ()=>{  
        const mockPet = {  
            name:"Orejitas",  
            specie:"Pez",  
            birthDate:"10-11-2022",  
        }  
    })  
})
```

Usando field y attach

Para poder enviar todos los campos, no solo los archivos, nos basaremos en el elemento **.field**

Sin embargo, cuando tengamos intención que colocar un archivo como elemento a enviar, se utilizará el elemento **.attach**.

Ergo, para poder enviar el objeto de mascota completo, incluyendo su imagen, lo haremos de la siguiente forma:

Recuerda que, al hacer un **attach**, este debe coincidir con el campo esperado por multer en su middleware **uploader.single**

```
const result = await requester.post('/api/pets/withimage')
  .field('name',mockPet.name)
  .field('specie',mockPet.specie)
  .field('birthDate',mockPet.birthDate)
  .attach('image','./test/coderDog.jpg');
```

Finalmente, procedemos a los expects

Algunos de los expects principales podrían ser los mostrados en la captura, sin embargo, queda bajo tu criterio qué elementos probar para dar por “finalizado” el test.

```
//Corroboramso que la petición haya resultado en OK
expect(result.status).to.be.eql(200);
//Corroboramso que el payload tenga un _id, indicando que se guardó en la BD
expect(result._body.payload).to.have.property('_id');
//Finalmente, corroboramos que la mascota guardada también tenga el campo image definido.
expect(result._body.payload.image).to.be.ok;
```

¡Enhorabuena!

¡Hemos conseguido realizar una estructura sólida de testing con diferentes contextos y diferentes retos a resolver!

💡 Recuerda tener contemplados siempre los tests en tu proyecto, son cruciales para un perfil laboral sólido en la empresa donde te desenvuelvas.

```
Testing Adoptme
  Test de mascotas
    ✓ El endpoint POST /api/pets debe crear una mascota correctamente (149ms)
    ✓ El endpoint POST /api/pets debe arrojar un error de status 400 si se envian valores incompletos
  Test avanzado
    ✓ Debe registrar correctamente a un usuario (215ms)
    ✓ Debe loguear correctamente al usuario Y DEVOLVER UNA COOKIE (139ms)
    ✓ Debe enviar la cookie que contiene el usuario y destrucutar éste correctamente
  Test uploads
    ✓ Debe poder crearse una mascota con la ruta de la imagen (85ms)

  6 passing (618ms)
```

¿Preguntas?

CODERHOUSE



Módulos de Testing para el proyecto “Adoptme”



ACTIVIDAD PRÁCTICA

Módulos de testing para proyecto final

Consigna

- ✓ Realizar módulos de testing para tu [proyecto principal](#), utilizando los módulos de mocha + chai + supertest.

Aspectos a incluir

- ✓ Se deben desarrollar los tests para todos los endpoints de:
 - Router de users.
 - Router de pets.
- ✓ NO desarrollar únicamente tests de status, la idea es trabajar lo mejor desarrollado posible las validaciones de testing

Formato

- ✓ link del repositorio en github sin node_modules

Sugerencias

- ✓ Ya que el testing lo desarrollarás tú, no hay una guía de test por leer. ¡Aplica tu mayor creatividad en tus pruebas!

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Tests de integración
- ✓ Tests con superTest
- ✓ Elementos avanzados de test

Opina y valora
esta clase

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser

- grabada

Clase 11. Testing y Escalabilidad Backend

Frameworks de desarrollo: Nest js (Parte I)

CODERHOUSE

Temario

10

Testing Avanzado

- ✓ Tests de integración
- ✓ Tests con supertest
- ✓ Testing de elementos avanzados

11

Frameworks de desarrollo: Nestjs (Parte I)

- ✓ Framework de desarrollo
- ✓ NESTjs
- ✓ Métodos principales en NESTjs

12

Frameworks de desarrollo: Nestjs (Parte II)

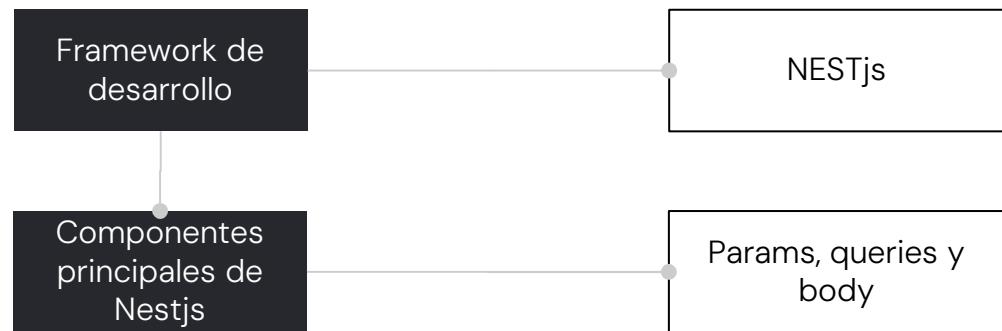
- ✓ Conexión de Nestjs a Mongo
- ✓ Autenticación con JWT

Objetivos de la clase

- Conocer qué es un framework de desarrollo
- Conocer qué es Nest js y la importancia de su uso
- Primeras implementaciones de Nestjs



MAPA DE CONCEPTOS



CLASE N°10

Glosario

Test unitario: Elemento más pequeño que puede ser testeado. La intención de las pruebas va dirigida al correcto funcionamiento de un módulo isolado.

Testing de integración: Tiene el objetivo de ver que los módulos funcionen en conjunto. Por ejemplo, módulo de Dao, Bcrypt.

Test funcional: Hace referencia a aplicar las integraciones de la misma manera que se realizaría un test de integración, sin embargo, enfocadas a cumplir una funcionalidad real.

Supertest: es una librería que nos permitirá ejecutar peticiones HTTP a nuestro servidor, para poder probar funcionalidades

CODERHOUSE

Framework de desarrollo

CODERHOUSE

¿Qué es un framework de desarrollo?

Los Frameworks son la principal herramienta generadora de **program scaffolds**. Un program scaffold te permite tener una estructura iniciada sobre algún modelo específico (un servidor, por ejemplo), sin embargo, esta estructura no estará finalizada y nosotros tendremos la posibilidad de ajustarla al modelo que nosotros necesitemos.



¿Por qué es diferente un framework de las librerías?

Una librería también cuenta con funcionalidades y tiene estructuras funcionales de apoyo.
Entonces... ¿En qué difiere?

El término clave es **inversión de control**. Éste **define los puntos del flujo en el que cada quién toma el control del aplicativo**.

Una librería tiene múltiples funciones. Sin embargo, en gran medida nosotros tomamos la decisión de en qué momento utilizarlo y sobre qué aplicarlo, es decir que nosotros decidimos el flujo de la funcionalidad.

Un **Framework** es un **marco de trabajo completo**, es decir, **tiene un flujo definido estructural**. Nos toca como desarrolladores unirnos a dicho flujo y comenzar a desarrollar a partir de ese marco.

Ventajas de utilizar un Framework

- ✓ **Facilita y agiliza el proceso de creación de un sistema web:** pueden levantarse sistemas completos de una manera sencilla si se cuenta con el conocimiento necesario.
- .
- ✓ **Estructura y organización de código predeterminada:** al tener una estructura predefinida al momento de levantar un proyecto, es más fácil familiarizarse con un esquema que ya se conoce, ya que habrá mucha información sobre ese esquema particular.

Ventajas de utilizar un Framework

- ✓ **Facilita el mantenimiento:** además de las buenas prácticas basadas en patrones que se utilizan, se aporta también a que múltiples desarrolladores puedan apoyarte al tener conocimiento similar de una estructura bien conocida
- ✓ **Permite concentrarnos sólo en el desafío de nuestro código:** al brindarnos un marco definido previamente solucionado y testeado, nos permite retomar el problema a partir de un avance, a diferencia de cuando tenemos que levantar una estructura desde 0, donde primero tendremos que resolver las raíces de **un problema genérico**, para después resolver el problema específico que deseamos.

Desventajas de utilizar un Framework

- ✓ **Curva de aprendizaje:** los frameworks son marcos no pensados para que se experimente con ellos, sino para que se aprenda a utilizarlos. Esto limita a los aprendices de dicho framework a tener que tomarse el tiempo para ir por el step by step de la curva de aprendizaje de un framework.
- ✓ **Si el framework no es minimalista, probablemente tengamos añadidas algunas funcionalidades que realmente no utilizaremos.** Esto no es necesariamente malo ya que en la realidad no suele haber una repercusión real en el performance, sin embargo, tampoco es bueno.

Desventajas de utilizar un Framework

- ✓ **Aprendizaje cerrado:** a pesar de compartir patrones y características, **manejar un framework significa aprender más que sólo una sintaxis nueva.** Se trata de una implementación de patrones diferente. Aprender otro framework significa en algunos casos, reaprender la implementación de algún patrón en una nueva sintaxis, o en el peor de los casos, aprender un flujo de trabajo completamente nuevo, lo cual desestabiliza una curva de aprendizaje.

Realmente no hay mucho que decir sobre los defectos de un framework, sin embargo, hay algo que sí tenemos que destacar...

👉 Usar un framework significa hacer una gran apuesta

El principal punto por el cual nos percatamos de un framework es por su **popularidad**, y muchas veces es una tecnología que no tocamos hasta que nos prometen que **el framework "x" es el futuro, y que "y" está muerto**. (Seguramente lo has escuchado más de una vez).

Pasa que después de un tiempo, nos damos cuenta, de que el dichoso framework "x" nunca fue "el futuro", además de darte cuenta que "y" en realidad nunca murió. Entonces utilizamos nuestro tiempo aprendiendo un framework que al final no fue laboralmente valioso, y descartamos a aquel que se mantuvo estable.

👉 Usar un framework significa hacer una gran apuesta

Hay que analizar a profundidad el framework que estamos por aprender. Podríamos volvernos expertos en una herramienta que nos asegurará trabajo durante el resto de nuestra carrera, o bien algo que pase “de moda” de la noche a la mañana.

¡Hay que analizar muy bien, para reconocer cuándo es necesario moverse a una nueva tecnología!

NETjs

CODERHOUSE

Nestjs

Es un framework que permite crear Server Side Applications, de una manera eficiente y escalable. Está desarrollado con una base sólida de **TypeScript** y brinda la posibilidad de seguir utilizando plain javascript.

Permite implementar paradigmas combinados de Programación Orientada a Objetos, Programación Funcional, y Programación Funcional reactiva.

Es un framework que está basado internamente en **Express**, y está pensado principalmente para construir aplicaciones monolíticas y microservicios.

Instalación y primer proyecto

Esta vez no utilizaremos esta implementación dentro de express, por lo tanto, haremos una instalación global a partir del comando:

```
npm i -g @nestjs/cli
```

Luego, para poder crear un nuevo proyecto, ejecutamos:

```
nest new primerProyectoNest
```

Nest nos preguntará qué package manager queremos utilizar, manejaremos el habitual: npm

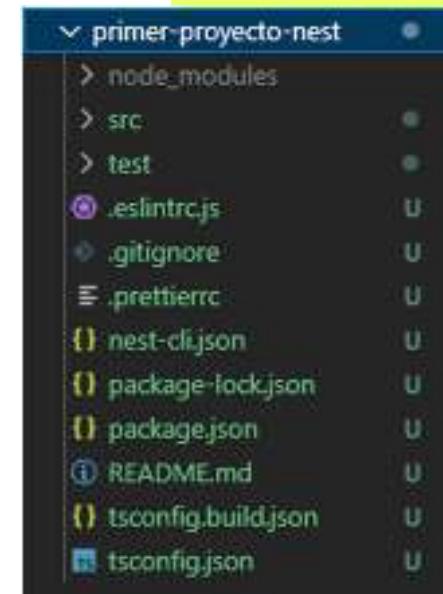
```
? Which package manager would you ❤️ to use? (Use arrow keys)
> npm
yarn
pnpm
```

Entendiendo la estructura del proyecto

CODERHOUSE

¿Qué generó Nestjs?

Comencemos entendiendo la estructura de los archivos externos a nuestra carpeta principal **src**



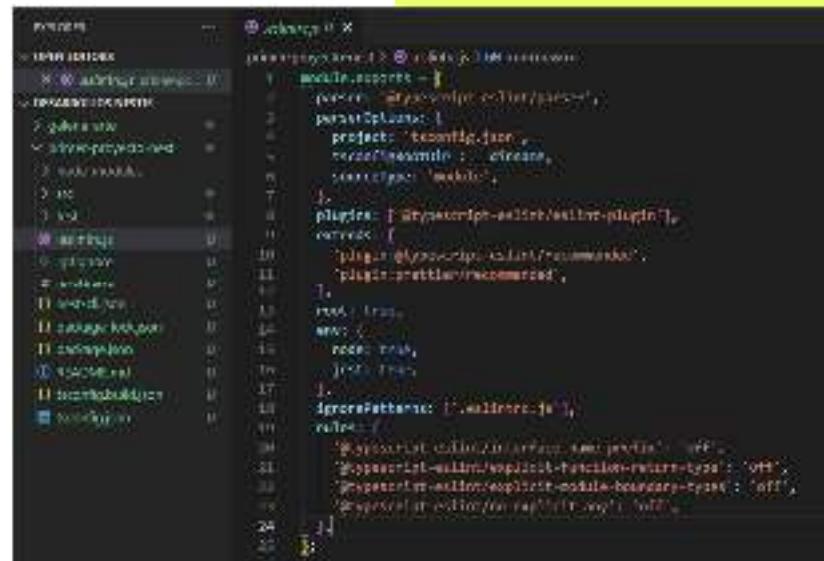
CODERHOUSE

¿Qué generó Nestjs?: ESLint

Primero está **ESLint**: el archivo que nos permitirá definir un conjunto de reglas para poder armar una guía de las prácticas de desarrollo que queremos que se cumplan.

Nota que estas reglas aparecen apagadas para no interferir “por defecto” con nuestro desarrollo.

Si quieras aprender a configurar tus propias reglas de desarrollo, puedes revisar la página oficial de ESLint [aquí](#)



The screenshot shows a code editor with two panes. The left pane displays the project structure of a NestJS application, including files like `src/app.module.ts`, `src/app-routing.module.ts`, `src/app.service.ts`, `src/app.component.ts`, `src/app.component.html`, `src/app.component.css`, `src/app.component.spec.ts`, `src/app.module.e2e.ts`, `src/app-routing.e2e.ts`, `src/app.service.e2e.ts`, `src/app.component.e2e.ts`, and `src/app.module.e2e.spec.ts`. The right pane shows the `eslint.config.js` file with the following content:

```
module.exports = {
  parser: "babel-eslint",
  parserOptions: {
    project: "teamig.json",
    tsconfigPath: "tsconfig",
    sourceType: "module"
  },
  plugins: ["@typescript-eslint/eslint-plugin"],
  extends: [
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  rules: {
    "no-unused-vars": "off",
    "no-restricted-globals": "off",
    "no-underscore-dangle": "off",
    "no-negated-condition": "off",
    "no-nested-ternary": "off",
    "no-dupe-class-members": "off",
    "no-dupe-keys": "off",
    "no-dupe-args": "off",
    "no-dupe-else-if": "off",
    "no-dupe-var": "off",
    "no-dupe-object-keys": "off",
    "no-dupe-require": "off",
    "no-dupe-import": "off",
    "no-dupe-imports": "off",
    "no-dupe-exports": "off",
    "no-dupe-classes": "off",
    "no-dupe-func": "off",
    "no-dupe-args": "off",
    "no-dupe-else-if": "off",
    "no-dupe-var": "off",
    "no-dupe-object-keys": "off",
    "no-dupe-require": "off",
    "no-dupe-import": "off",
    "no-dupe-imports": "off",
    "no-dupe-exports": "off",
    "no-dupe-classes": "off",
    "no-dupe-func": "off"
  }
};
```

¿Qué generó Nestjs?: Prettier

Luego tenemos **Prettier**, el cual nos permitirá definir unas reglas de estilo para el proyecto, nota que vienen activadas las singleQuotes y se colocará automáticamente la trailingComma.

Si quieres aprender a definir tus propias reglas de estilos, puedes investigarlo en el sitio oficial de Prettier [aquí](#)

The screenshot shows the VS Code interface. On the left, the Explorer pane displays the project structure: DESARROLLOS, primer-proyecto-nest, node_modules, src, test, eslintrc.js, .gitignore, .prettierrc (which is selected), nest-dts.json, package-lock.json, package.json, README.md, tsconfig.build.json, and tsconfig.json. On the right, the Editor pane shows the contents of the .prettierrc file:

```
1: {  
2:   "singleQuote": true,  
3:   "trailingComma": "all"  
4: }
```

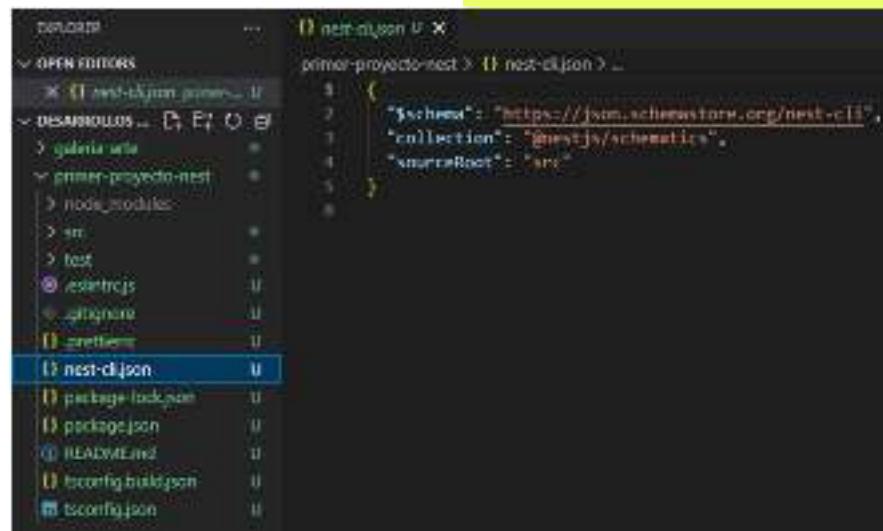
CODERHOUSE

¿Qué generó Nestjs?: nest-cli.json

Este archivo nos permite tener contempladas múltiples configuraciones, una de éstas es para poder generar un *dist* final.

Un ejemplo de una configuración diferente es:

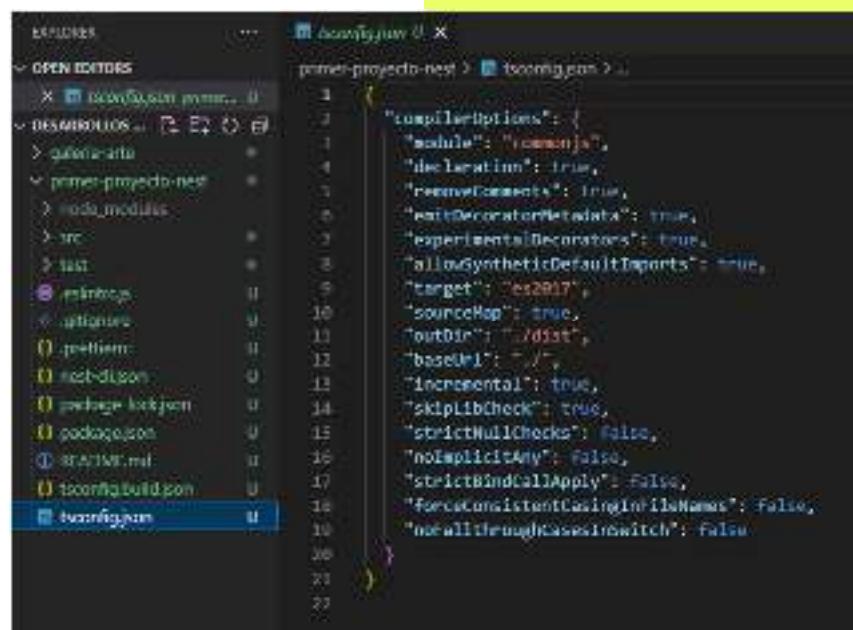
```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src",
  "compilerOptions": {
    "assets": [
      { "include": "**/*.yml", "watchAssets": true }
    ],
    "plugins": ["@nestjs/swagger/plugin"]
  }
}
```



¿Qué generó Nestjs?: tsconfig.json

En caso de que trabajemos con Typescript, genera también un archivo para poder configurar la transpilación hacia el JavaScript final.

Recuerda que Typescript es un **superSet de Javascript**, puedes profundizar sobre este lenguaje y su sintaxis [aquí](#).



The screenshot shows a code editor with the file 'tsconfig.json' open. The left pane displays the project structure in the Explorer view, including 'node_modules', 'src', 'nestjs', 'actions', 'entities', 'interfaces', 'package-lock.json', 'package.json', 'README.md', and 'tsconfig.build.json'. The right pane shows the contents of the 'tsconfig.json' file:

```
compilerOptions": {  
  "module": "commonjs",  
  "declaration": true,  
  "removeComments": true,  
  "emitDecoratorMetadata": true,  
  "experimentalDecorators": true,  
  "allowSyntheticDefaultImports": true,  
  "target": "es2017",  
  "sourceMap": true,  
  "outDir": "/dist",  
  "baseUrl": "./",  
  "incremental": true,  
  "skipLibCheck": true,  
  "strictNullChecks": false,  
  "noImplicitAny": false,  
  "strictBindCallApply": false,  
  "forceConsistentCasingInFileNames": false,  
  "noFallthroughCasesInSwitch": false  
}
```

¡Importante!

Al ser un superset de JavaScript, Typescript te permitirá trabajar con las mismas sintaxis y operaciones, de manera que el uso de este nuevo lenguaje es bastante intuitivo. Si no lo conocías, **te recomendamos que comiences a aprenderlo en paralelo.**

Una última cosa... ¡Vamos a echar a andar el proyecto!

Una vez que terminamos el tour por los archivos externos a la carpeta `src`, vamos a levantar el servidor con el típico comando `npm start`.

Este buscará el script en `package.json` y ejecutará el comando:

```
format: "prettier --write",  
"start": "nest start",  
"format": "prettier --write"
```

Tendremos entonces nuestro aplicativo escuchando en el puerto 3000

Hello World!

CODERHOUSE

Componentes principales de Nestjs

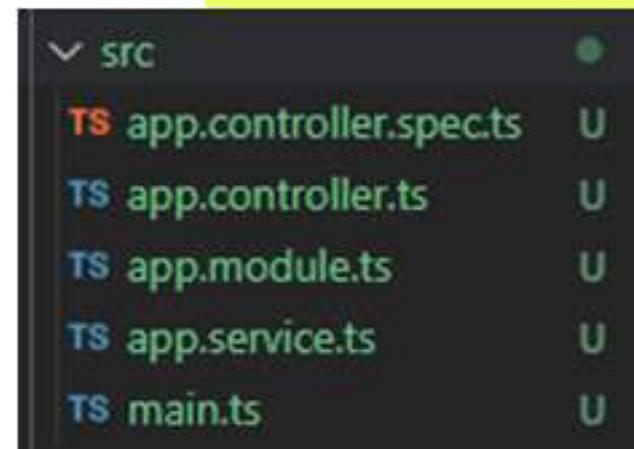
CODERHOUSE

Componentes generales de la arquitectura Nestjs

Esta vez vamos a abrir la carpeta src y vamos a revisar qué ha generado.

Podemos separarlo en 5 elementos principales:

- ✓ spec
- ✓ controller
- ✓ module
- ✓ service
- ✓ main



```
src
  TS app.controller.spec.ts
  TS app.controller.ts
  TS app.module.ts
  TS app.service.ts
  TS main.ts
```

ANALIZANDO LA CARPETA src:

main

Está el corazón de nuestro aplicativo.

Nota que el **método NestFactory crea el módulo principal a partir de un módulo**. Este contendrá entonces las funcionalidades principales de nuestro aplicativo.

El servidor además escucha en el puerto 3000. ¡Ahí se aloja actualmente nuestro "Hello World!"

Ahora, el motor principal es bastante fácil de entender, es prácticamente nuestro **app.js** habitual, ¿pero qué es eso de crear un módulo?

A screenshot of a code editor showing the main.ts file in the src folder. The file contains the following code:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}

bootstrap();
```

The code uses the NestFactory.create method to create an application module, which is defined in the AppModule. The application then listens on port 3000.

ANALIZANDO LA CARPETA src:

module

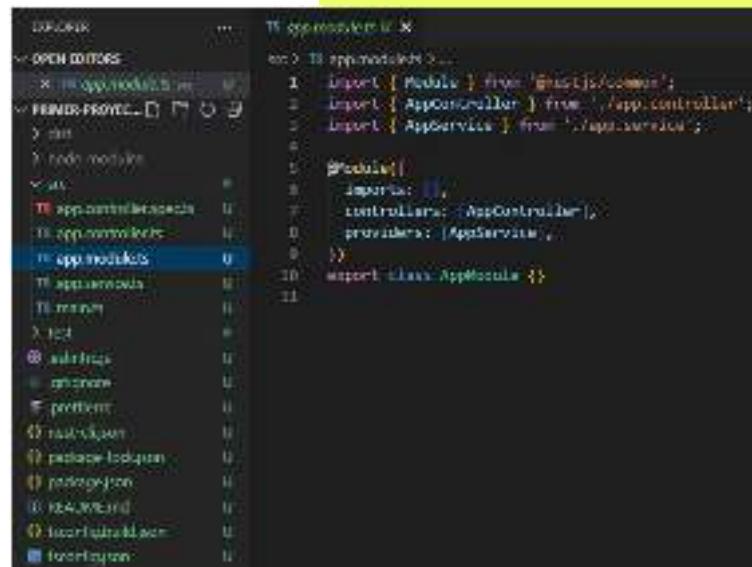
Aquí deja de ser tan familiar el código. ¿Qué es esto que estamos viendo?

Estos @ que estás viendo son llamados **decoradores**.

El decorador es un elemento instaurado en Javascript y Typescript, cuya funcionalidad sigue la lógica del **patrón de diseño decorador**. ¿Y cómo funciona?

Básicamente es una **herencia**, pero esta vez no es aplicada sólo a clases habituales, sino que también se aplican a funciones.

Entendiendo este concepto, podemos decir que con el decorador @Module, estamos indicando que lo que se encuentra entre paréntesis será tratado como un módulo.



```
src/app/app.module.ts
1 import { NgModule } from '@angular/core';
2 import { AppComponent } from './app.component';
3 import { AppController } from './app.controller';
4 import { AppService } from './app.service';
5
6 @Module({
7   imports: [],
8   controllers: [AppController],
9   providers: [AppService],
10 })
11 export class AppModule {}
```

¡Más sobre el módulo!

Además, notamos que hay tres aspectos importantes a analizar:

- ✓ **imports**: Son utilizados para obtener los módulos que requerimos como dependencias.
- ✓ **controllers**: Podemos entenderlo como los controladores que ya conocemos dentro del mundo de Express. Éstos consumirán los servicios que viajan desde los providers. La diferencia está en la función que cumple cada uno.
- ✓ **providers**: Contienen la verdadera definición de lo que se declara por los controllers, es decir, contiene las operaciones reales que tienen el funcionamiento interno de la operación.

The screenshot shows a code editor with two tabs open. The left tab is titled 'app.module.ts' and contains the following TypeScript code:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

The right tab shows the file system structure of the project:

- src
- app
- controller
- entity
- module
- provider
- service
- util

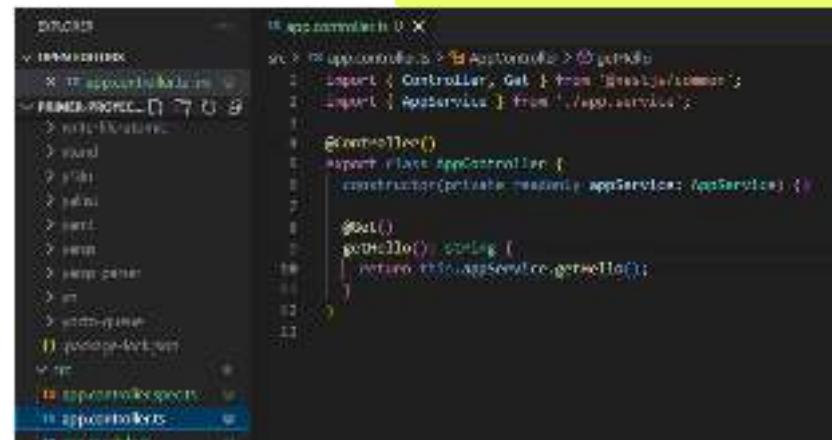
ANALIZANDO LA CARPETA src:

controller

Nuevamente nos encontramos múltiples decoradores, sólo recordando, éstos harán que la clase Appcontroller cuente con las funcionalidades del Controller, y que el método getHello funcione como un método Get.

Notemos cómo el método getHello estará ejecutando la implementación del servicio appService. De cierta forma se despreocupa del funcionamiento interno del servicio (como lo hacemos en Express).

El return del controller será el equivalente **a hacer una respuesta al cliente** (res)



```
src > app > appController.ts > AppController > getHello()
1 import { Controller, Get } from 'nestjs/common';
2 import { AppService } from './app.service';
3
4 @Controller()
5 export class AppController {
6   constructor(private readonly appService: AppService) {}
7
8   @Get()
9   getHello(): string {
10     return this.appService.getHello();
11   }
12 }
```

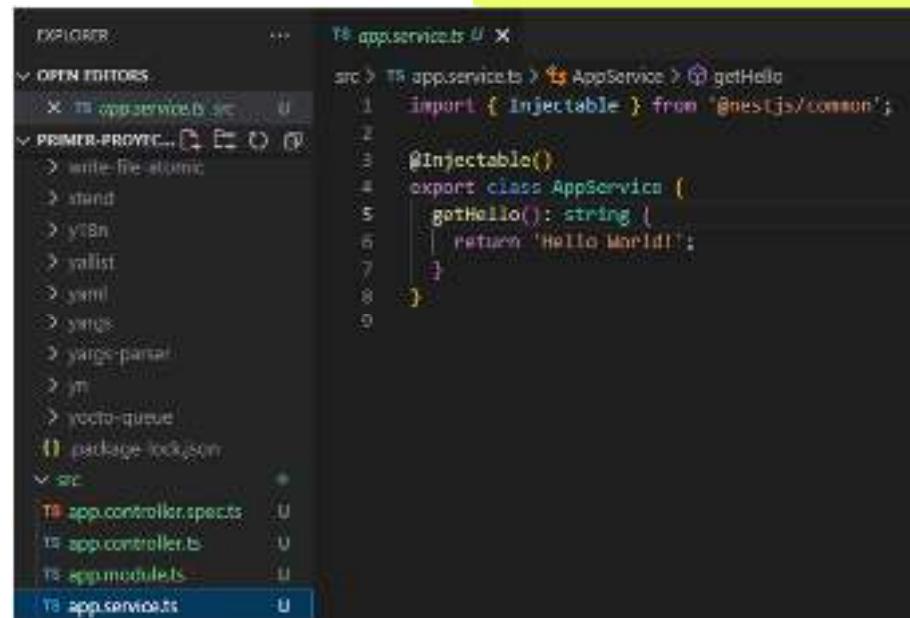
ANALIZANDO LA CARPETA src:

service

Las implementaciones de servicios que permitan traer información irán en la parte de service. Éste nos servirá para poder implementar múltiples operaciones y conectar con módulos de acceso a datos, para pasarlo al controlador.

El **return** que podemos ver en el método getHello, pasa el dato al controlador para que éste responda al cliente.

Hello World!



```
src > TS app.service.ts > AppService > getHello
1 import { Injectable } from '@nestjs/common';
2
3 @Injectable()
4 export class AppService {
5   getHello(): string {
6     return 'Hello World!';
7   }
8 }
```

CODERHOUSE

Hay mucho por explorar

Analizar un framework en profundidad es un proceso largo que requiere de mucha paciencia. Iremos profundizando sobre los conceptos a medida que los vayamos utilizando.

Ahora, destinaremos el resto de la clase a implementar cada uno de los métodos principales de un servidor en este framework.

Realizaremos un GET, POST, PUT, DELETE, utilizando una persistencia en memoria.



Break

¡10 minutos y volvemos!

CODERHOUSE

Métodos principales en Nestjs

CODERHOUSE

Creando nuestro propio módulo

Vamos a recrear un módulo pensado específicamente para trabajar con usuarios

Para ello, utilizaremos una funcionalidad bastante interesante: los **resources**.

Antes de explicarlos, vamos a correr el comando:

```
nest g resource Users
```

Primero, nos preguntará a partir de qué modelo queremos crear dicha estructura (Mira la cantidad de compatibilidades que tiene)

```
? What transport layer do you use?  
> REST API  
GraphQL (code first)  
GraphQL (schema first)  
Microservice (non-HTTP)  
WebSockets
```

Seleccionaremos REST API. Y Nest comenzará a hacer lo suyo: Darnos un marco de trabajo para ese recurso en particular.

¡Creamos el módulo, y algo más!

Notamos que al solicitar el recurso, no sólo estamos generando un módulo, sino que además nos acaba de crear:

- ✓ El servicio para ese módulo.
- ✓ Un controlador para ese módulo
- ✓ Entidades representativas de dicho módulo
- ✓ DTOs para dicho módulo
- ✓ Testings de servicios y controllers para ese módulo.

Y por último, si vamos a nuestro app.module:

Nota cómo automáticamente agregó nuestro módulo creado a los imports del módulo principal



The image shows a file explorer window with a 'users' folder expanded. Inside 'users' are subfolders 'dto' and 'entities', and files 'create-user.dto.ts', 'update-user.dto.ts', 'users.controller.spec.ts', 'users.controller.ts', 'users.module.ts', 'users.service.spec.ts', and 'users.service.ts'. Below the file explorer is a code editor displaying the 'app.module.ts' file. The code defines an 'AppModule' class with imports for 'Module', 'AppController', 'AppService', and 'UsersModule'. It then uses the '@Module()' decorator to define the module, specifying 'imports: [UsersModule]', 'controllers: [AppController]', and 'providers: [AppService]'. Finally, it exports the 'AppModule' class.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';

@Module({
  imports: [UsersModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

¿Y el ruteo?

Vayamos al archivo users.controller.ts que nos generó el recurso.

Nota cómo en el decorador Controller tiene una string dentro 'users', esto significa que, automáticamente creamos también un router/controller e incluso viene incluido con los métodos principales como Posts, Gets, Patch y Delete

PD: La diferencia entre un Patch y un Put radica en la completitud de los datos: PUT está pensado para actualizar todo, PATCH sólo para un fragmento de la información.



```
1 // src/controllers/user.ts
2 import { Controller, get, post, body, patch, Param, Delete } from 'src/decorators';
3 import { UserEntity } from 'src/entities/user.entity';
4 import { UserService } from 'src/services/user';
5
6 @Controller('users')
7 export class UserController {
8   constructor(private readonly userService: UserService) {}
9
10   @Get()
11   async findAll(@Param('id') id: string): Promise<UserEntity> {
12     return await this.userService.findById(id);
13   }
14
15   @Post()
16   async create(@Body() user: UserEntity): Promise<UserEntity> {
17     return await this.userService.create(user);
18   }
19
20   @Patch()
21   async update(@Param('id') id: string, @Body() user: UserEntity): Promise<UserEntity> {
22     return await this.userService.update(id, user);
23   }
24
25   @Delete()
26   async remove(@Param('id') id: string): Promise<UserEntity> {
27     return await this.userService.remove(id);
28   }
29 }
```

Vamos a modificar un poco los servicios

Abriremos los servicios de usuarios (que también generó Nestjs) y vamos a crear nuestra persistencia (un array de la entidad User definido en el archivo user.entity.ts).

La persistencia la crearemos desde el constructor e inicializamos a los usuarios que devolveremos, crearemos, modificaremos y eliminaremos.

Modificaremos el findAll para que devuelva dicho array (básicamente sí, es acceder a nuestra fuente de datos)

Este *return* llegará al controller, así que tenemos que modificar el controller también para modificar el cuerpo de la respuesta, para que se sienta más como “lo que hacíamos en Express”

```
18 userservice.ts U: ✘ 18 userentity.ts U
src/users/userservice.ts > 18 userservice
1 ✓ import { Injectable } from '@nestjs/common';
2 import { CreateUserDto } from './dto/create-user.dto';
3 import { UpdateUserDto } from './dto/update-user.dto';
4 import { User } from './entities/user.entity';
5
6
7 @Injectable()
8 export class UserService {
9   users: Array<User>;
10
11   constructor() {
12     this.users = [];
13   }
14
15   create(createUserDto: CreateUserDto) {
16     return 'This action creates a user';
17   }
18
19   findAll() {
20     return this.users;
21   }
22
23   findOne(id: number) {
24     const user = this.users.find((user) => user.id === id);
25     if (!user) {
26       throw new Error(`User with id ${id} not found`);
27     }
28     return user;
29   }
30
31   update(id: number, updateUserDto: UpdateUserDto) {
32     const user = this.users.find((user) => user.id === id);
33     if (!user) {
34       throw new Error(`User with id ${id} not found`);
35     }
36     Object.assign(user, updateUserDto);
37     return user;
38   }
39
40   remove(id: number) {
41     const index = this.users.findIndex((user) => user.id === id);
42     if (index === -1) {
43       throw new Error(`User with id ${id} not found`);
44     }
45     this.users.splice(index, 1);
46     return `User with id ${id} removed`;
47   }
48 }
```

User Entity

En la carpeta User se nos creó una carpeta adicional "entities", en ésta, colocaremos la entidad usuario, para poder utilizarla en las múltiples operaciones.

Consistirá en una clase que podremos tomar como modelo de datos principal para los movimientos que realicemos con nuestras fuentes de datos.

(Esto porque es importante especificar un tipo de dato para Typescript, entonces así contaremos con el tipo de dato "User".

```
export class User {  
    id:Number;  
    first_name:string;  
    last_name:string;  
    email:string;  
    password:string;  
    avatar:string;  
}
```

Modificando la respuesta del controller

Antes

```
14
15  @Get()
16  findAll() []
17    return this.userService.findAll();
18  }
19
```

[]

Después

```
14
15  @Get()
16  findAll() {
17    const users = this.userService.findAll();
18    return {status:"success",users}
19  }
20
```

{"status":"success","users":[]}

CODERHOUSE

Params, queries y body

CODERHOUSE

Parametrizando

```
@Get('/:id')
findOne(@Param('id') id: string) {
    return this.userService.findOne(+id);
}

@Patch('/:id')
update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.userService.update(+id, updateUserDto);
}

@Delete('/:id')
remove(@Param('id') id: string) {
    return this.userService.remove(+id);
}
```

El id, que se está pasando como parámetro en la función **findOne**, **update** o **remove**, también es relacionado con un decorador, en este caso **@Param**. Éste permitirá que se reciba de manera directa como parámetro de un método http y no necesariamente como un parámetro de una función cualquiera.

Validar un parámetro

```
@Get('/:id')
findOne(@Param('id') id: string) {
    if(isNaN(+id)) throw new HttpException('Invalid param',HttpStatus.BAD_REQUEST);
    return this.usersService.findOne(+id);
}
```

Para poder hacer una validación, podemos preguntar si el parámetro es numérico. En caso de que no lo sea, arrojar un error de Http a partir de un **HttpException** (La excepción Http y el diccionario de status codes viene incluido en el framework)

Manejo de queries

Modificaremos la función findAll para limitar el número de usuarios a recibir, para esto, si se manda un query param *limit*, limitaremos según el número de usuarios indicados, caso contrario devolverá por defecto 5 usuarios.

Utilizaremos un nuevo decorador `@Query`:

- ✓ Si usamos Query sin parámetros, tomará todos los parámetros enviados.
- ✓ También podemos limitar exactamente qué valores de los query params tomar, por ejemplo `@Query('parametro') alias`

```
@Get()
findAll(@Query() query) {
  const {limit} = query;
  console.log(limit);
  const users = this.userService.findAll();
  return {status:"success",users}
}
```

```
@Get()
findAll(@Query('limit') limit) []
  console.log(limit);
  const users = this.userService.findAll();
  return {status:"success",users}
]
```

Finalmente, trabajando con body

Finalmente, para poder manejar con parámetros, Nest también nos ha proporcionado un decorador específico para poder leer bodies, al final, nos ayudará a tener un control más específico. Nota cómo podemos recibir ya un esquema predefinido, pero la validación de campos podrá ser similar.

```
@Post()
create(@Body() createUserDto: CreateUserDto) {
    if(!createUserDto.first_name||!createUserDto.email||!createUserDto.password) throw new HttpException('Incomplete values',HttpStatus.BAD_REQUEST)
    return this.usersService.create(createUserDto);
}
```

Todo en uno

¿Recuerdas lo maravilloso que era poder contar con el req.body, req.params y req.query, todo en un solo objeto sin declararlos uno a uno?

Nest nos proporciona también un decorador **Request**, el cual contará con todo lo necesario para poder acceder como lo hacíamos antes.

¡Ahora tenemos todos los decoradores esenciales para manejar nuestros requests correctamente!

```
@Post('/:b')
probarRequest(@Request() req){
    console.log(req.query);
    console.log(req.params);
    console.log(req.body);
    return "¡Todo en un objeto!"
}
```



Finalizando el servicio

Duración: 5-10 min



ACTIVIDAD EN CLASE

Finalizando los métodos

Con base en la estructura previamente realizada.

Finalizar todos los servicios ligados al controlador para tener una funcionalidad completa con el módulo de usuarios.

Se debe poder crear un usuario, buscar un usuario, ver todos los usuarios, actualizar un usuario, y borrar un usuario. Todo se seguirá manejando con persistencia en memoria



Para pensar

Hasta este punto, ¿Cuál consideras que es el punto de mantener una estructura tan rigurosa en la forma de hacer los requests de la entidad users?

¿Consideras que la estructura planteada ayuda o afecta a la forma de trabajar en el backend?



Hands on lab

Crearemos un nuevo módulo products.

Éste se deberá ejecutar bajo el comando nest g resource.

¿De qué manera?

- ✓ Modificaremos la lógica del findAll y el create para que se pueda crear un producto y ver los productos en memoria.
- ✓ No es necesario realizar los demás métodos.
- ✓ Modificaremos la entidad Product, para que ésta pueda fungir como el modelo representativo de la persistencia de productos.

Tiempo estimado: **10/15 minutos**

CODERHOUSE

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Concepto e instalación de Nestjs
- ✓ Arquitectura y archivos de Nestjs
- ✓ Módulos, controladores y servicios
- ✓ Entidades, DTOs y decoradores.
- ✓ GET, POST, PATCH, DELETE en Nestjs

Opina y valora
esta clase

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 12. Testing y Escalabilidad Backend

Frameworks de desarrollo: Nest js (Parte II)

CODERHOUSE

Temario

11

Frameworks de desarrollo: Nestjs (Parte I)

- ✓ Framework de desarrollo
- ✓ NESTjs
- ✓ Métodos principales en NESTjs

12

Frameworks de desarrollo: Nestjs (Parte II)

- ✓ Conexión de Nestjs a Mongo
- ✓ Autenticación con JWT

13

Práctica integradora

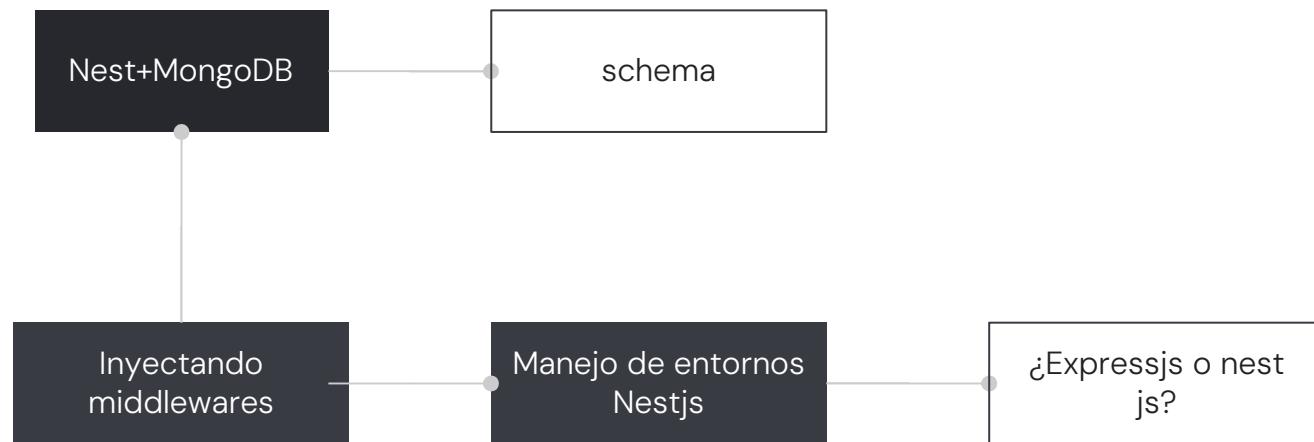
- ✓ Skills
- ✓ Práctica integradora

Objetivos de la clase

- **Utilizar** la arquitectura de Nestjs para poder conectar con la base de datos de Mongo a partir de mongoose
- **Trabajar** con un modelo de autenticación con jwt



MAPA DE CONCEPTOS



CLASE N°11

Glosario

Frameworks: son herramientas generadoras de program scaffolds.

Nestjs: es un framework que permite crear Server Side Applications, de una manera eficiente y escalable.

Program Scaffold: permite tener una estructura iniciada sobre algún modelo específico.

Nestjs + MongoDB

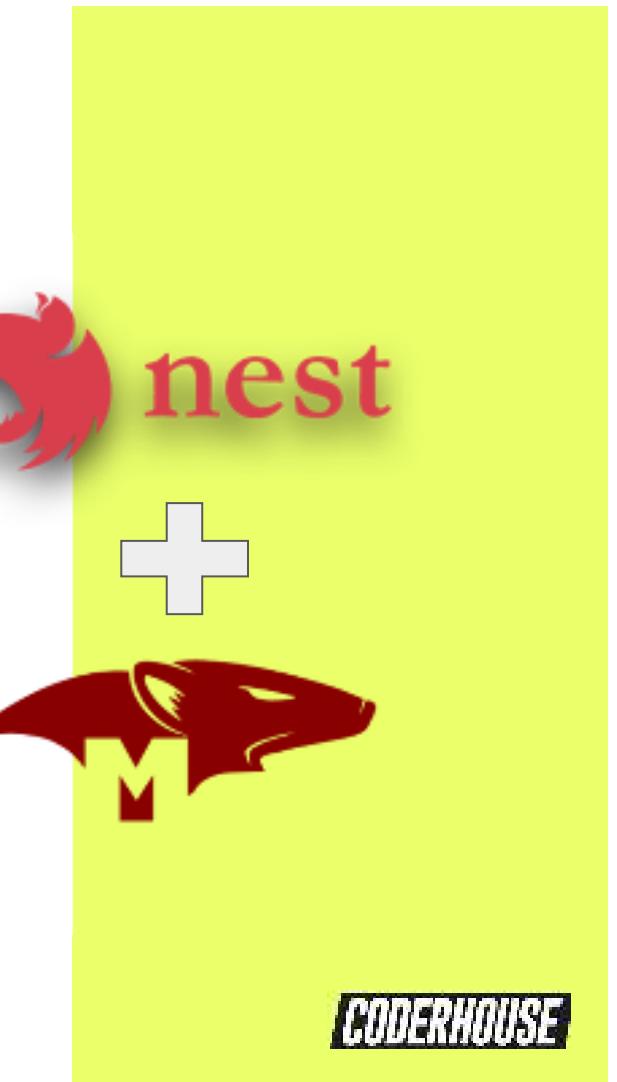
CODERHOUSE

¿Cómo funciona mongoose en Nest?

Recordemos que Mongoose nos ayudará a estructurar esquemas para poder manejar nuestra base de datos, de manera que podremos seguir trabajando con el modelo de persistencia predilecto del curso.

Nestjs proporciona un módulo interno de compatibilidad con mongoose para poder obtener su máximo potencial.

Sin embargo, recordemos que debido al tipado, Nest utilizará cierta sintaxis y tipos de datos particulares para poder operar con una compatibilidad completa con los datos de mongoose.



APROXIMACIÓN AL PROCESO

Instalación

Antes que nada, levantaremos un proyecto nuevo de Nest con **nest new nombre-del-proyecto**

Una vez teniendo el proyecto, vamos a instalar mongoose y también la integración de Nest con Mongoose:

```
npm i @nestjs/mongoose mongoose
```

Una vez instalado, podemos conectarlo con el módulo principal: **app.module**



APROXIMACIÓN AL PROCESO

Usar mongoose en app.module

Para poder comenzar a utilizarlo, necesitamos establecer nuestra primera conexión, para ello importaremos en app.module, un módulo de Mongoose que apunte a **nuestra típica base de Mongo Atlas**

Importamos en el módulo de app un **MongooseModule.forRoot**. Esto nos servirá para establecer la conexión desde el inicio de la aplicación.

```
TS app.module.ts U X
coderbase > src > ts app.module.ts > AppModule
  1 import { Module } from '@nestjs/common';
  2 import { MongooseModule } from '@nestjs/mongoose';
  3 import { AppController } from './app.controller';
  4 import { AppService } from './app.service';
  5
  6 @Module({
  7   imports: [MongooseModule.forRoot('TU URL DE ATLAS AQUÍ')],
  8   controllers: [AppController],
  9   providers: [AppService],
 10 })
 11 export class AppModule {}
```

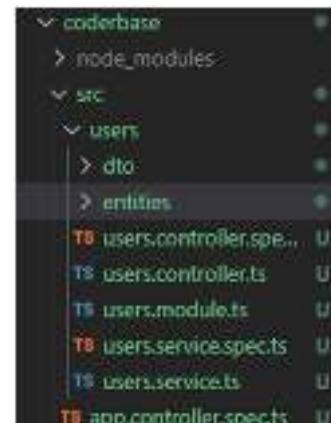
APROXIMACIÓN AL PROCESO

Volvamos a declarar a nuestros usuarios

Ejecutaremos nuevamente la funcionalidad de resource para crear nuestra carpeta de usuarios:

```
nest g resource users
```

El comando creará la carpeta elemental que creamos en la clase pasada:





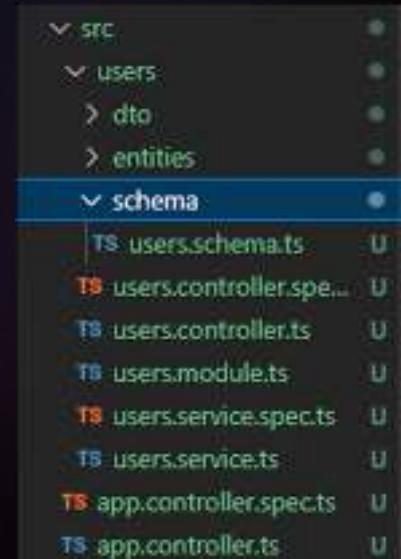
PARA RECORDAR

Pensemos en Mongo

Estamos trabajando con mongoose. **Ello significa que los datos que estamos por generar deberán tener un schema para poder generar el modelo de mongoose.**

Sin embargo, la construcción de este schema será algo diferente a lo que estamos acostumbrados.

¡Veamos qué tanto cambia el hacer un schema para mongoose en Nest!



CODERHOUSE

¡Analicemos un schema!

Primero la línea 4: esta línea nos permitirá construir un tipo de dato `UsersDocument`, el cual vendrá construido por el **schema** de abajo, e indicando si este será mezclado con la configuración de un **Documento hidratado** o no.

Recuerda que el documento hidratado hace referencia a que los resultados devueltos por la base sean devueltos como **Instancias de documento de mongo**, lo cual significa que cuenta con múltiples funcionalidades adicionales de mongo.

Si queremos objetos planos de JS. usamos `LeanDocument<User>`

```
users.schema.ts •
coderbase > src > users > schema > users.schema.ts > User
1 import {Prop, Schema, SchemaFactory} from '@nestjs/mongoose';
2 import {HydratedDocument} from 'mongoose';
3
4 export type UsersDocument = HydratedDocument<User>
5
6 @Schema()
7 export class User {
8   @Prop()
9   first_name:string;
10
11   @Prop()
12   last_name:string;
13
14   @Prop()
15   email: string;
16
17 }
18
```

¡Analicemos un schema!

Además, notemos que tenemos dos nuevos decorators:
@Schema y @Prop.

Recuerda que con mongoose, podemos poner ciertas configuraciones a cada prop: Podemos hacer lo mismo como argumentos del decorator.

Y por último, la línea especial:

```
export const UserSchema = SchemaFactory.createForClass(User);
```

Al final del archivo, colocaremos la creación del schema final, esto a partir de una dependencia especial de Nest/mongoose llamada **SchemaFactory**

```
@Schema()
export class User {
  @Prop({ required:true })
  first_name:string;

  @Prop()
  last_name:string;

  @Prop({ required:true, unique:true })
  email: string;
}
```

CODERHOUSE

Conectar el schema de mongoose a nuestro contexto de usuarios

Cuando generamos el recurso de usuarios, este creó un **users.module.ts**, que cuenta con un contexto interno de controladores y providers.

Sin embargo, el hecho de crearlo solo con estos dos componentes no significa que no podamos utilizar la tercera característica principal de un módulo: **los imports**.

Como mencionamos la clase pasada, los imports permiten importar otros imports. Por ejemplo, app.module, importa el módulo users.module, de la misma forma entonces, el users.module podrá importar un módulo de Mongoose que nos permitirá darle el contexto del schema que recién creamos.

Ejemplos

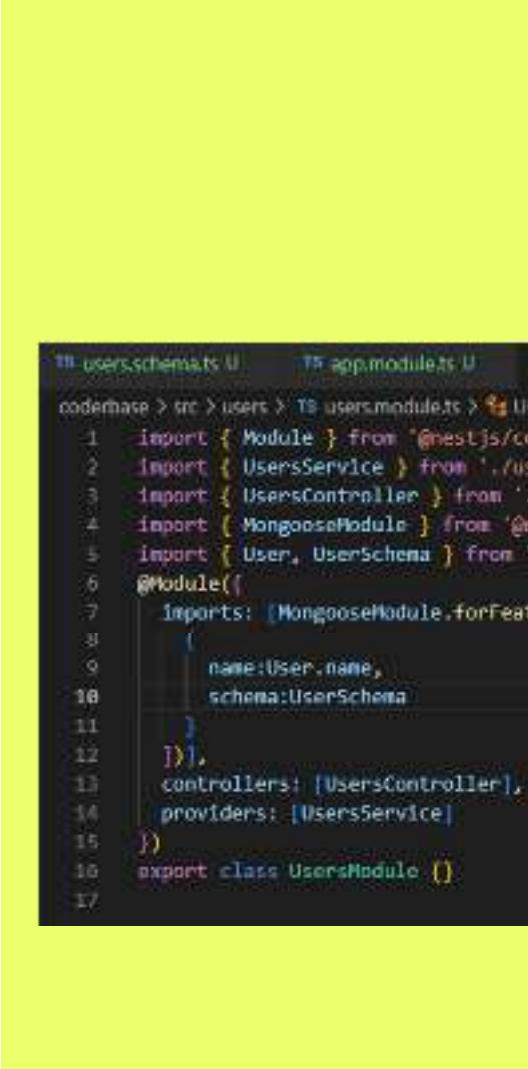
app.module importando a los users

```
coderbase > src > TS app.module.ts > App
  6
  7   @Module({
  8     imports: [UsersModule, MongooseModule],
  9     controllers: [AppController],
10     providers: [AppService],
11   })
12   export class AppModule {}
```

users.module importando MongooseModule

```
@Module({
  imports: [MongooseModule.forFeature([
    {
      name: User.name,
      schema: UserSchema
    }
  ]),
  controllers: [UsersController],
  providers: [UsersService]
})
export class UsersModule {}
```

Sobre el import de Mongoose.Module



```
users-schema.ts | app.module.ts | users.module.ts ✘
codernode > src > users > TS users.module.ts > UsersModule
  1 import { Module } from '@nestjs/common';
  2 import { UserService } from './users.service';
  3 import { UsersController } from './users.controller';
  4 import { MongooseModule } from '@nestjs/mongoose';
  5 import { User, UserSchema } from './schema/users.schema';
  6 @Module({
  7   imports: [MongooseModule.forFeature([
  8     {
  9       name: User.name,
 10       schema: UserSchema
 11     }
 12   ]),
 13   controllers: [UsersController],
 14   providers: [UserService]
 15 ])
 16 export class UsersModule {}
```

Para entender lo que hicimos, debemos tener en cuenta:

- ✓ MongooseModule ya es un módulo incluido en la dependencia de nestjs/mongoose. Necesita que especifiquemos todos los modelos que vamos a usar.
- ✓ Luego, muy similar a un **collection, schema**, que aplicábamos para generar nuestros modelos de mongoose en Express, mandamos el **name** y el **schema** generado del archivo de users.schema previo.

Ahora todo el módulo de usuarios **sabe cómo utilizar** Usuarios generados por mongoose. El resto es aplicarlo en los servicios.

Inyectando el modelo en el servicio

Vamos a inyectar un decorador
@InjectModel

Como ya tenemos el contexto de quiénes son nuestros usuarios (contexto que pasamos desde el app.module). Sólo hacemos la inyección del nombre del modelo del usuario.

Luego, declararemos una variable (privada) que represente el modelo que vamos a utilizar en los servicios, nota que el modelo utiliza el tipo de dato **UsersDocument** que creamos en el archivo del schema.

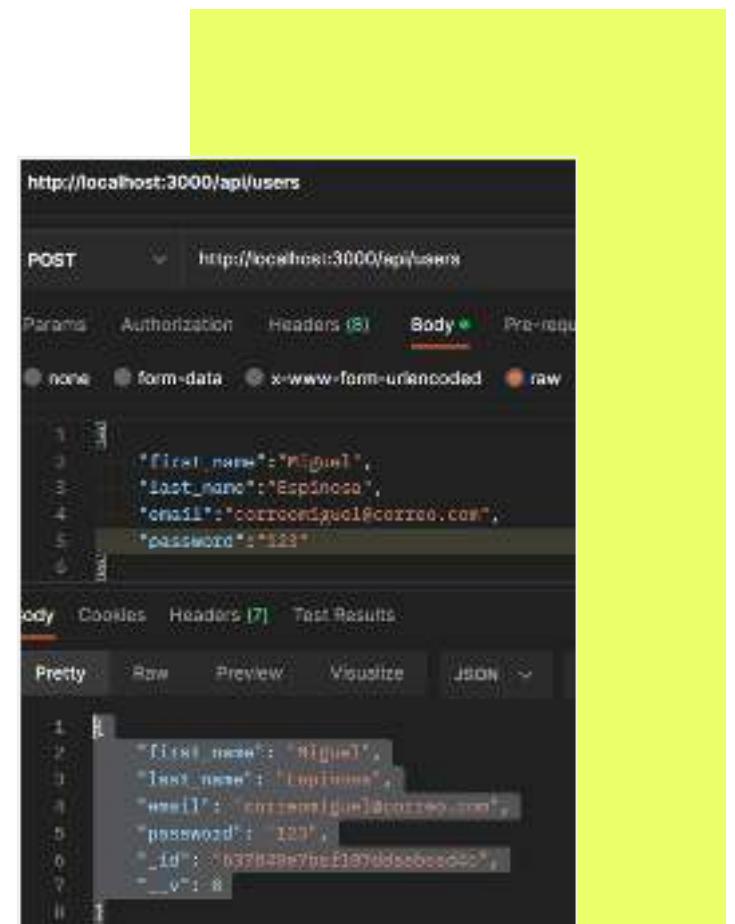
```
coderbase > sm > users > lib > userService.ts > 8 UserService > 13 create
1 import { Injectable } from '@nestjs/common';
2 import { InjectModel } from '@nestjs/mongoose';
3 import { CreateUserServiceDto } from './dto/create-user.dto';
4 import { UpdateUserServiceDto } from './dto/update-user.dto';
5 import { User, UserDocument } from './schemas/users.schema';
6 import { Model } from 'mongoose';
7 @Injectable()
8 export class UserService {
9
10   constructor(@InjectModel(User.name) private userModel: Model<UserDocument>) {}
11
12   create(createUserServiceDto: CreateUserServiceDto): Promise<User> {
13     return this.userModel.create(createUserServiceDto);
14   }
15 }
```

Conexión + contexto + variable...

Fuimos conectando, archivo por archivo, los diferentes nodos que se necesitan para que un módulo pueda tener control total de un modelo de Mongo. Entonces, procedamos a hacer una inserción:

```
create(createUserDto: CreateUserDto) {
  return this.usersModel.create(createUserDto);
}
```

Nota que ahora podemos utilizar **this.usersModel** en cualquiera de nuestros archivos, y las operaciones de mongoose, a las que tanto estamos acostumbrados, permanecen idénticas.



CODERHOUSE



Completando CRUD de mongo

Duración: 5-10min

CODERHOUSE



ACTIVIDAD EN CLASE

Completando CRUD de mongo

A partir de la conexión realizada con Mongo.

Completar el servicio de usuarios para seguir implementando el modelo de usuarios y finalizar las operaciones de CRUD

No es necesario implementar ningún DTO o entidad, puede devolverse en raw data.



Injectando middlewares

CODERHOUSE

Recordemos que Nestjs está basado en express

Al estar basado en Express, éste conserva algunos de los patrones que éste aplica de manera tan distintiva.

Es por ello que tendremos la posibilidad de setear middlewares con el fin de colocar todos estos puntos intermedios entre la petición y el endpoint.

En este caso particular, cada middleware representará una clase completa. Por ello, crearemos una carpeta específica para el desarrollo de los middlewares a utilizar.



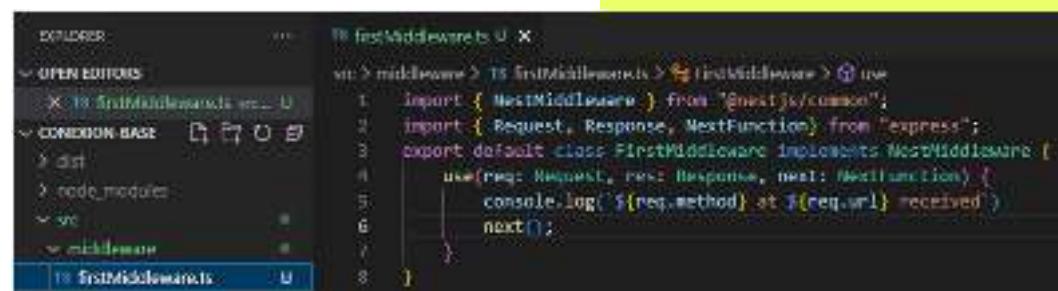
CODERHOUSE

Estructura de un middleware en Nest js

El archivo que se genera en esta carpeta lo nombraremos **firstMiddleware.ts**.

Luego, este archivo contendrá una clase que será la combinación de los elementos básicos de Express (Request, Response, Next), además de una funcionalidad extendida que implementa de NestMiddleware.

El objetivo de este middleware será mostrar el método y la ruta que se está visitando.



```
ts FirstMiddleware.ts (1) [selected]
src > middleware > FirstMiddleware.ts
1 import { NestMiddleware } from '@nestjs/common';
2 import { Request, Response, NextFunction } from 'express';
3 export default class FirstMiddleware implements NestMiddleware {
4   use(req: Request, res: Response, next: NextFunction) {
5     console.log(`${req.method} at ${req.url} received`);
6     next();
7   }
8 }
```

¿Cómo conectarlo con nuestros módulos?

Para que un módulo sea consciente de un middleware, es importante que éste cuente con un **consumidor de Middleware**, el cual es una funcionalidad interna del framework.

Nuestro módulo AppModule, deberá implementar NestModule para que pueda utilizar de manera interna el **configure**, una función que permitirá setear el consumidor que queramos.

```
src > TS app.module.ts (1) AppModule > ⚡ configure
1 import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
2 import { ConfigModule, ConfigService } from '@nestjs/config';
3 import { MongooseModule } from '@nestjs/mongoose';
4 import { AppController } from './app.controller';
5 import { AppService } from './app.service';
6 import { UsersModule } from './users/users.module';
7
8 @Module({
9   imports: [UsersModule, ConfigModule.forRoot(), MongooseModule.forRootAsync({
10     imports:[ConfigModule],
11     inject:[ConfigService],
12     useFactory: async(config:ConfigService) => ({{
13       url: config.get<string>('MONGO_URL')
14     })
15   })
16   controllers: [AppController],
17   providers: [AppService],
18 })
19 export class AppModule implements NestModule {
20   configure(consumer: MiddlewareConsumer) {
21     consumer
22     .apply()
23   }
24 }
```

CODERHOUSE

Aplicación del middleware al consumer

Al hacer `consumer.apply`, indicamos qué middleware será el que utilizaremos para fungir como intermediario de dicho módulo. Sin embargo, a diferencia de Express, éste nos solicita indicar directamente cuáles serán las rutas que están autorizadas para utilizar dicho middleware

Al colocar `{path: '*', method: RequestMethod.All}` indicamos que este middleware funcionará para todas las rutas, para todos los métodos, siempre que necesitemos alguna ruta más específica, o algún método más específico, podemos hacerlo en un objeto aparte.

```
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(FirstMiddleware).forRoutes({path:'*',method:RequestMethod.ALL})
  }
}
```

Middleware en acción

Ahora, cada vez que mandemos a llamar a nuestro servidor, independientemente de la ruta o el método, siempre podremos visualizar un log indicando la siguiente información:

```
GET at /users received at 13:17:55
```

Este sólo es un middleware informativo, sin embargo, al igual que cualquier otro middleware desarrollado en el curso, puedes utilizarlo para hacer validaciones, realizar decoración del request, inicializar alguna otra variable, etc.



Break

¡10 minutos y volvemos!

CODERHOUSE

Manejo de entornos de Nestjs

CODERHOUSE

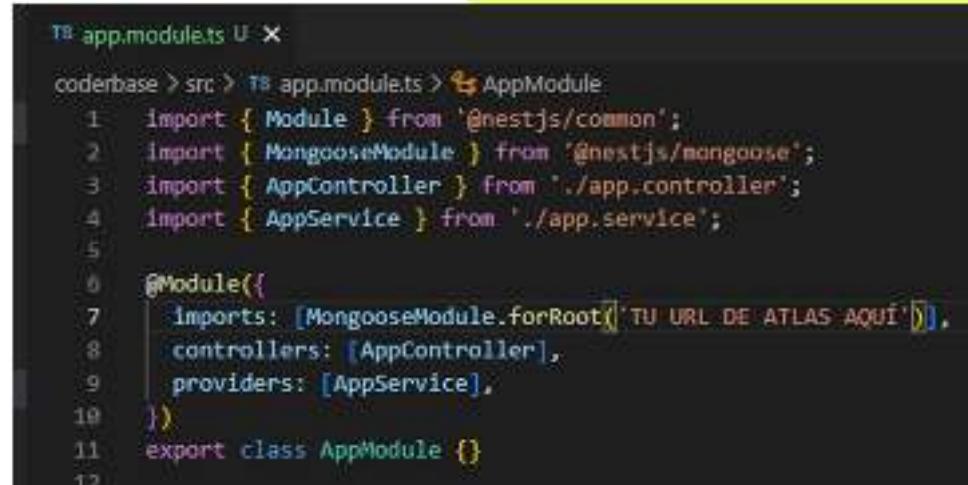
Tenemos problemas con nuestra conexión

En la primera parte de esta clase, pudimos hacer una conexión con nuestra base de datos, sin embargo, hay un tema importante que tenemos que voltear a ver:

¿Es realmente la forma más óptima de poder realizar nuestra configuración de base de datos?

Notemos que la base tiene expuesta su ruta en el método forRoot del MongooseModule.

Sabemos que este elemento debería estar oculto por un entorno para solo tener que mandarlo a llamar, y nuestra ruta no se encuentre expuesta.



```
codenbase > src > app.module.ts > AppModule
  1 import { Module } from '@nestjs/common';
  2 import { MongooseModule } from '@nestjs/mongoose';
  3 import { AppController } from './app.controller';
  4 import { AppService } from './app.service';

  5 @Module({
  6   imports: [MongooseModule.forRoot('TU URL DE ATLAS AQUÍ')],
  7   controllers: [AppController],
  8   providers: [AppService],
  9 })
 10 }
 11 export class AppModule {}
```

¿Cómo funciona un entorno de Nest?

Muy similar a lo que nosotros teníamos en Express como una carpeta de config. Nestjs también cuenta con un configModule.

Este contará con los elementos necesarios para acceder a variables de entorno, el Módulo permitirá utilizar la variable config, para poder extraer la variable de entorno de éste cuando sea necesario.

Además, la configuración al final es un Módulo. Necesitaremos que los demás módulos que vayan a utilizar dicho entorno, sean conscientes de éste.

Para esto, vamos a colocar en los imports del módulo, el módulo de configuración.

Habilitando la configuración en nuestro proyecto

Lo primero será instalar la configuración en nestjs, para eso utilizaremos el comando **npm install @nestjs/config**

```
npm install @nestjs/config
```

Ahora, con el módulo ya instalado, vamos a importarlo en nuestro app.module.ts de la siguiente forma:

```
import { Module } from '@nestjs/common';
import { ConfigModule, ConfigService } from '@nestjs/config';
```

Estos dos elementos nos servirán para configurar el módulo de app:

- ✓ ConfigModule se utiliza en los imports
- ✓ ConfigService se injecta para usarse como servicio.

Analizando los cambios del módulo App (1)

El primer cambio importante que vemos es en la propiedad de imports: Se utiliza un ConfigModule.forRoot(), esto permite inicializar el contexto del módulo de configuración para que sea conocido y contemplado por el aplicativo principal y en los módulos internos.

Luego, tenemos el cambio en MongooseModule, donde esta vez estamos haciendo una inicialización dinámica. Nota cómo el configModule también se pasa al import de MongooseModule, así también como el ConfigService se inyecta para que pueda ser utilizado dentro de MongooseModule.

```
@Module({
  imports: [UsersModule,ConfigModule.forRoot(), MongooseModule.forRootAsync({
    imports:[ConfigModule],
    inject:[ConfigService],
    useFactory: async(config:ConfigService) => {
      uri: config.get<string>('MONGO_URL')
    }
  }),
  controllers: [AppController],
  providers: [AppService],
})
```

Analizando los cambios del módulo App (2)

Finalmente, tenemos un elemento `useFactory`, el cual es utilizado para crear un provider de manera dinámica, la razón de hacerlo así es que podremos inicializarlo ya tomando la url de conexión directamente de config.

Config extiende de **ConfigService**, es por ello que aplicando un `get` podemos apreciar como se accede a una variable de entorno desde Nest. **config.get<string>('Nombre de la variable')** buscará el archivo `.env` y tomará la variable con ese nombre, en caso de no encontrarla, devolverá `undefined`.

Solo queda colocar en el `.env` la string de conexión de mongo.

```
@Module({
  imports: [UsersModule, ConfigModule.forRoot(), MongooseModule.forRootAsync({
    imports: [ConfigModule],
    inject: [ConfigService],
    useFactory: async(config: ConfigService) => ({
      uri: config.get<string>('MONGO_URL')
    })
  ]),
  controllers: [AppController],
  providers: [AppService],
})
```

¿Cómo se utilizan variables en otros módulos?

Como comentamos, para que un módulo funcione para otros módulos, basta con importar el módulo en el arreglo de imports del module, como se ve en la imagen. ☺

```
ts users.module.ts 1/ X  
src > users > ts users.module.ts > ...  
1 import { Module } from '@nestjs/common';  
2 import { UsersController } from './users.controller';  
3 import { MongooseModule } from '@nestjs/mongoose';  
4 import { User, UserSchema } from './schemas/user.schema';  
5 import { ConfigModule } from '@nestjs/config';  
6  
7 @Module({  
8   imports:[MongooseModule.forRoot(  
9     {  
10       name:User.name,  
11       schema:UserSchema  
12     }  
13   ),  
14   ConfigModule],  
15   controllers: [UsersController],  
16   providers: [UserService]  
17 })  
18 export class UsersModule {}
```

Además, el servicio será inyectado en el controlador

Cuando declaremos un controlador, dentro del constructor de dicho controlador podemos injectar el servicio de configuración.

Automáticamente, podremos utilizarlo en cualquier endpoint que deseemos.

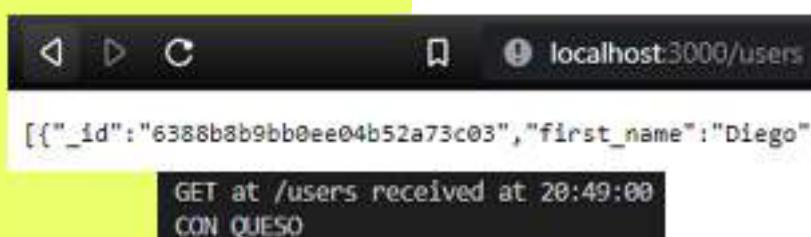
```
import { ConfigService } from '@nestjs/config';
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UserService, private config: ConfigService) {}
```

```
❶ .env ❷ X  
❶ .env  
❷ 1 MONGO_URL='TU URL DE MONGO AQUÍ'  
❷ 2 PAPA = CON_QUESO
```

Así, teniendo un archivo .env, declaramos los siguientes campos.

```
@Get()  
findAll() []  
console.log(this.config.get<string>('PAPA'))  
return this.userService.findAll();  
[]
```

En la ruta base del método get de /users, utilizamos el config para obtener la variable PAPA



Notamos que al visitar endpoint, se muestra por consola el valor de la variable de entorno.

¿Expressjs o Nestjs?

CODERHOUSE

¿Qué elegir para mi trayectoria?

A pesar de que estas clases has visto Nestjs como un framework alternativo de Expressjs, sin embargo, ésto no significa que tengas que tomarlo como el reemplazo directo y olvidar lo visto en el curso.

La principal diferencia entre Nestjs y Expressjs está en su carácter dogmático

¿A qué se refiere ésto? Expressjs no es dogmático, ya que te permite tener un flujo de libertad de configuración en múltiples elementos del mismo, sin que éste se vea afectado en su core principal, sin embargo, Nestjs te brinda ya las herramientas que necesitas para realizar una tarea, evitando así que configures demasiadas cosas externas al framework

Principales puntos a tratar

Expressjs

- ✓ No es dogmático
- ✓ No tiene una arquitectura definida.
- ✓ Altamente popular y con alto soporte de la comunidad.
- ✓ Bastante útil para aprender a realizar configuraciones, ya que tenemos la libertad de hacerlo.

Nestjs

- ✓ Es dogmático
- ✓ Tiene una arquitectura definida para ajustarnos a ella
- ✓ En crecimiento lento, pero constante, aún no con tanto soporte de la comunidad en comparativa con express
- ✓ Bastante útil para aprender a seguir un marco definido y con elementos ya hechos, pues la configuración en mayor medida no está hecha por nosotros

Ambas alternativas están en el tope

A pesar de que Nestjs es menos popular que Expressjs, éste está ganando amplia popularidad y está posicionado como el segundo mejor framework de desarrollo de nodejs.

De momento, Expressjs es mucho más sólido en términos de apoyo de usuarios, documentación, soluciones, etc.

Te recomendamos ampliamente que lleves un crecimiento paralelo de trabajo con ambas soluciones, ya que de ambas se pueden aprender elementos diferentes que te servirán para tu trayectoria como desarrollador.





Recuerda estar en constante actualización

Las diapositivas están armadas acorde con las estadísticas de su momento. ¡Siempre hay que mantenerse en constante actualización y corroborar hacia dónde se inclina la balanza!

Al final, una vez terminado el curso, recuerda que el mayor responsable de tu portafolio al mundo siempre serás tú, y tus mejores herramientas para el mundo laboral son las tecnologías por las que apuestes

CODERHOUSE

¿Preguntas?

CODERHOUSE

¡Atención!

La próxima clase será una **Práctica Integradora**. Te recomendamos prepararte con la siguiente guía de temas



[Guía de skills](#)

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Conexión de bases de datos con Nestjs
- ✓ Manejo de middlewares con Nestjs
- ✓ Manejo de entornos de Nestjs
- ✓ Entidades, DTOs y decoradores.
- ✓ GET, POST, PATCH, DELETE en Nestjs

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 13. Testing y Escalabilidad Backend

Práctica integradora

CODERHOUSE

Temario

12

Frameworks de desarrollo: Nestjs (Parte II)

- ✓ Conexión de Nestjs a Mongo
- ✓ Autenticación con JWT

13

Práctica Integradora

- ✓ Skills
- ✓ Práctica integradora

14

Proceso principal del servidor + Global & Child process

- ✓ Despliegue de nuestro aplicativo
- ✓ Configuración de pipeline en Railway.app

Objetivos de la clase

- Hacer una integración práctica de todos los conceptos vistos hasta el momento, bajo el desarrollo de un proyecto paralelo a nuestro proyecto final.

¿Cómo organizar la clase?

Estructura general de la clase	Tiempo de dedicación	Enfoques
Skills	10 min	<ul style="list-style-type: none">✓ No detenerse a explicar cada skill✓ Utilízalo como motivador, no como stopper.
Práctica integradora	1hr 20 min (contemplando break a tu elección)	<ul style="list-style-type: none">✓ Recuerda que estás retomando un proyecto de práctica integradora previo. No pierdas tiempo repasando cosas de la práctica integradora anterior.✓ Esta práctica integradora supone un caso muy particular, y es que es de carácter más relajado, ya que las cosas a integrar son bastante sencillas, cosas como logging y documentación, haciendo que el resto de los temas sea bastante subjetivo según sea el caso.
Presentación del desafío	10 min	
Espacio de dudas y consultas	15 min	<ul style="list-style-type: none">✓ Puedes traer dudas con respecto a contenidos vistos en el curso o bien relacionadas con el terreno profesional.

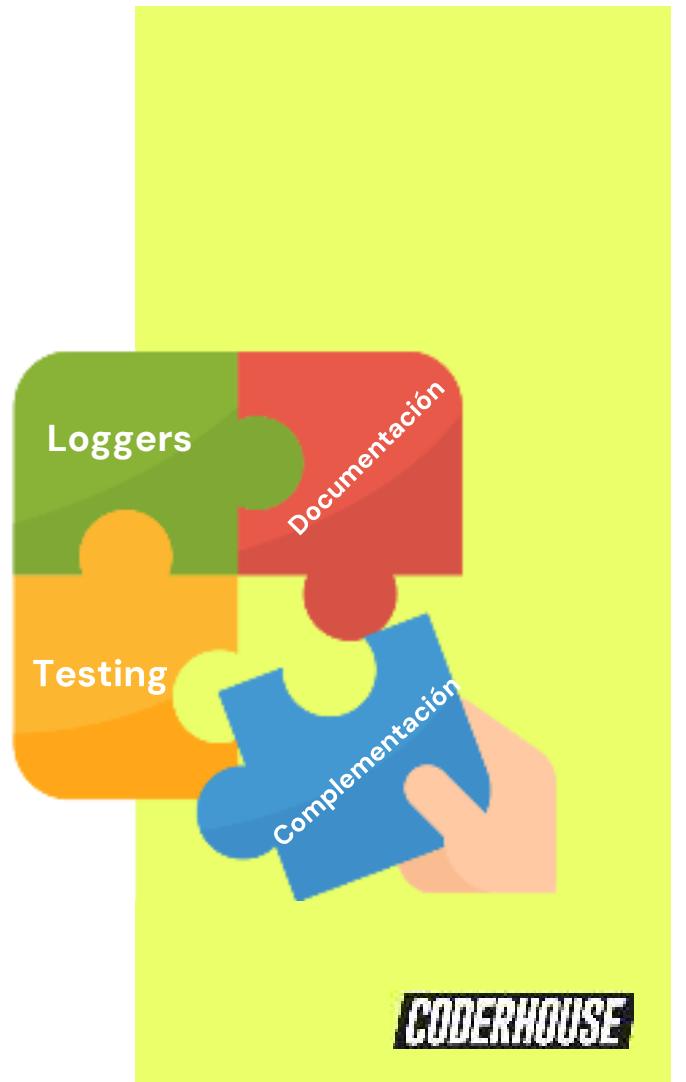
¿Qué estamos por ver?

CODERHOUSE

Práctica integradora

La mejor forma de repasar los temas vistos hasta el momento, es hacer un repaso integrado de todos los elementos.

Si bien es correcto repasar el código parte por parte, es importante que como desarrolladores comencemos a trabajar en nuestra **lógica de integración**, es decir, tenemos que tener siempre contemplado el cómo vamos a juntar todo lo aprendido, para tener un proyecto sólido



CODERHOUSE

Elementos a Integrar

CODERHOUSE

Elementos a integrar

En el desarrollo de esta práctica de integración, repasarás y conectarás los siguientes conceptos:

- ✓ Logging
- ✓ Documentación
- ✓ Complementación

**Skills para esta
práctica integradora**

CODERHOUSE

Skills para Logging

- ✓ Comprender la importancia de utilizar un logger
- ✓ Comprender el uso de Winston Logger y aplicarlo
- ✓ Entender los diferentes tipos de transportes
- ✓ Entender sobre los niveles de logging
- ✓ Configurar nuestros propios niveles de logging



CODERHOUSE

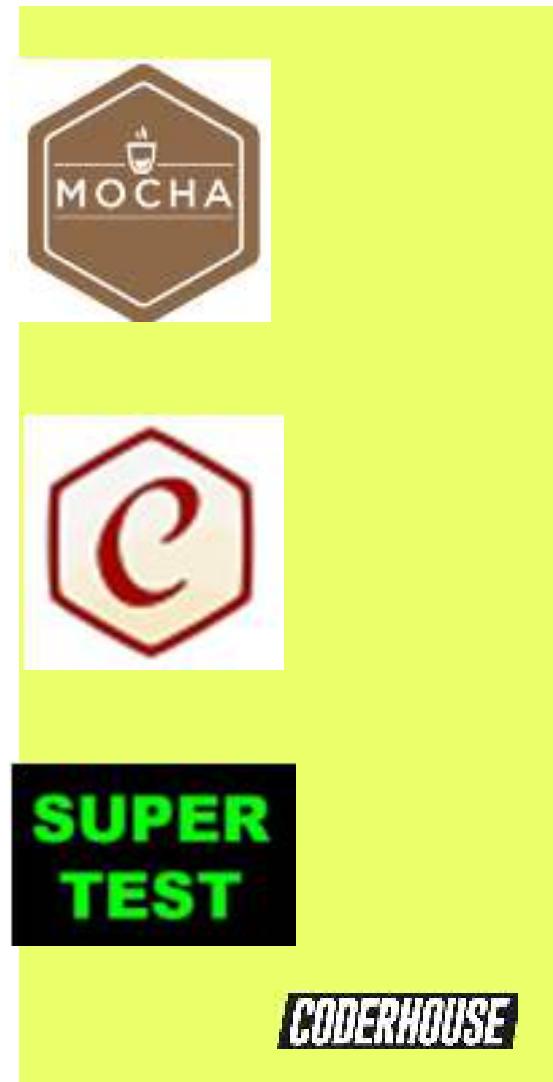


Skills para documentación

- ✓ Comprender la importancia de documentar
- ✓ Comprender el uso de Swagger para documentación
- ✓ Realizar la Swaggerización por archivos de cada Módulo
- ✓ Comprender sobre los elementos que compone un módulo Swaggerizado (schemas, inputs, requestBodies, responses, etc)

Skills para testing

- ✓ Comprender sobre módulos de Testing
- ✓ Conocimientos de realización de Testing unitario
- ✓ Conocimiento sobre Test de integración
- ✓ Uso de Mocha
- ✓ Uso de Chai
- ✓ Uso de SuperTest



Importante

La práctica hace al maestro... si consideras que aún necesitas practicar algún aspecto de los skills mencionados, siempre puedes repasar las clases previas. Sin embargo, esto no significa que no puedas avanzar.

¡Esta práctica integradora te ayudará a sentirte más seguro en estos temas!

Desarrollo de esta práctica integradora



1

Aplicación de un logger que utilice múltiples transportes

2

Documentación elemental del proyecto en cuanto a sus elementos principales

3

Testeo de funcionalidades principales de los módulos por separado e integrados.

¡Comenzamos!

CODERHOUSE



Práctica de integración sobre tu proyecto principal

CODERHOUSE



ACTIVIDAD PRÁCTICA INTEGRADORA

Consigna

Con base en el [proyecto principal](#) que venimos desarrollando, toca solidificar algunos procesos.

Aspectos a incluir

- ✓ Modificar el modelo de User para que cuente con una nueva propiedad “documents” el cual será un array que contenga los objetos con las siguientes propiedades
 - name: String (Nombre del documento).
 - reference: String (link al documento).

No es necesario crear un nuevo modelo de Mongoose para éste.
- ✓ Además, agregar una propiedad al usuario llamada “last_connection”, la cual deberá modificarse cada vez que el usuario realice un proceso de login y logout



ACTIVIDAD PRÁCTICA INTEGRADORA

Aspectos a incluir

- ✓ Crear un endpoint en el router de usuarios **api/users/:uid/documents** con el método POST que permita subir uno o múltiples archivos y actualizar el atributo “documents” del usuario en cuestión. Utilizar el middleware de Multer para poder recibir los documentos que se carguen en el proyecto.
- ✓ El middleware de multer deberá estar modificado para que pueda guardar en diferentes carpetas los diferentes archivos que se suban.
 - Si se sube una imagen de una mascota, deberá guardarlo en una carpeta **pets**, mientras que ahora al cargar un documento, multer los guardará en una carpeta **documents**.
- ✓ Desarrollar los tests funcionales para los endpoints de **api/sessions/register** y **api/sessions/login** utilizando los módulos de mocha, chai y supertest.



ACTIVIDAD PRÁCTICA INTEGRADORA

Formato

- ✓ Link al repositorio de GitHub con el proyecto completo (no incluir node_modules).

Sugerencias

- ✓ Para el uso de Multer, se puede tomar de referencia el endpoint **api/pets/withimage** que ya utiliza Multer para subir las imágenes de las mascotas.

**¿Dudas, preguntas,
consultas?**

CODERHOUSE

Muchas gracias.

CODERHOUSE

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 14. Testing y Escalabilidad Backend

Product Cloud: Despliegue de nuestra aplicación.

CODERHOUSE

Temario

13

Cuarta Práctica Integradora

- ✓ Skills
- ✓ Práctica integradora

14

Product Cloud: Despliegue de nuestro aplicativo

- ✓ Despliegue de nuestro aplicativo
- ✓ Configuración de pipeline en Railway.app

CODERHOUSE

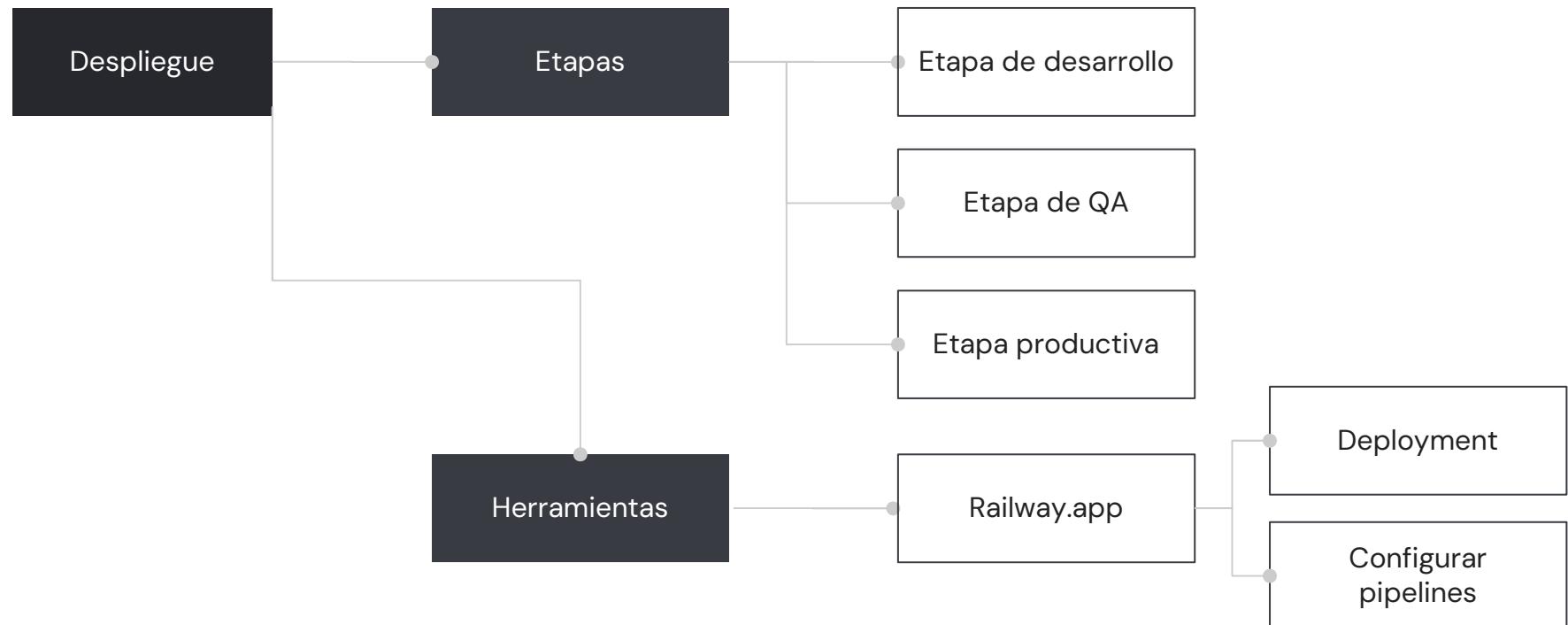
Objetivos de la clase



- Comprender sobre el proceso de deploy de un proyecto
- Manejar distintas etapas de desarrollo de aplicación



MAPA DE CONCEPTOS



Despliegue de nuestro aplicativo

CODERHOUSE

¿Nuestro aplicativo está listo para ser desplegado?

¿Cuándo podemos decir que nuestro aplicativo está finalizado? La realidad es que, dentro del mundo del desarrollo web, no podemos esperar a “terminar” un aplicativo para poder comenzar a desplegarlo.

La liberación de un proyecto es algo más complejo de lo que parece, y ocupa diferentes etapas que dependen de la empresa en la cual se está trabajando.

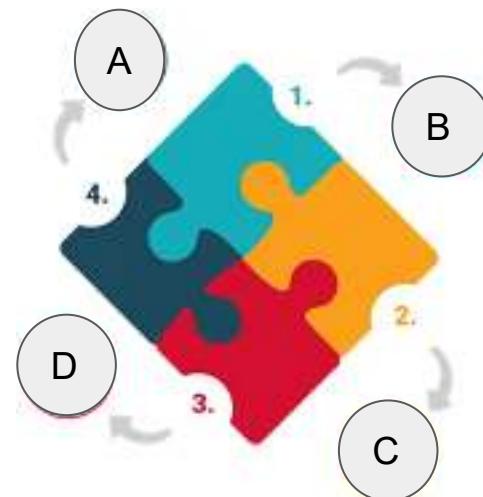


CODERHOUSE

Etapas en el desarrollo de aplicaciones

Cuando se trata de sacar un proyecto al mercado, debe pasar por un conjunto de etapas, sobre las cuales se evalúan diferentes elementos para considerarse listo para el cliente. Las etapas varían enormemente según la forma de trabajo de cada empresa, por lo tanto, colocaremos en este punto las tres etapas principales por las cuales pasa un aplicativo:

- ✓ Etapa de desarrollo
- ✓ Etapa de QA
- ✓ Etapa productiva



Etapa de desarrollo

Esta etapa está pensada para el flujo libre del desarrollador en cuanto a los cambios del aplicativo que éste pueda realizar.

En esta primera etapa, el desarrollador realiza y prueba su código en un **entorno demasiado amigable**. Se lo considera amigable, porque se desarrolla en el contexto que tiene en dicho momento el desarrollador, además de pasar por las pruebas elementales que el desarrollador pensó.

Sin embargo, cuando el desarrollador está listo para decir adiós a su aplicativo, tiene que enviarlo a las siguientes etapas



CODERHOUSE

Etapa de QA

Una vez que el desarrollador considera que su aplicativo está listo para ser presentado, este “**pasa**” la app a la siguiente etapa, que es QA.

Quality Assurance está pensado para poder ejecutar la feature del aplicativo que el desarrollador recién acaba de entregar. En su mayoría se tratan de **pruebas de blackbox** (pruebas sin saber del código, solo funcionalidad directa).

Si el aplicativo tiene el visto bueno por parte del equipo de QA, éste estará listo para salir a ser utilizado por el cliente. Es en este momento final cuando se dice que se hace un **despliegue a productivo**.

En caso de que no se dé el visto bueno, tendrá que regresar a etapa de desarrollo para hacer los respectivos cambios y presentar nuevamente.

Etapa productiva

Cuando QA autoriza liberación, entonces el aplicativo **“pasa”** a la etapa productiva. Éste es el punto final en el que el aplicativo vive en el mundo real, se enfrenta a clientes y comportamientos reales, con datos reales, con elementos cambiantes y con formas inusuales de usarse.

Cuando el aplicativo ha pasado por las etapas previas, el rango de error debería ser enormemente reducido, sin embargo, **el cliente es capaz de causar errores indescriptibles en el uso real.**

Es por ello que, siempre que se requiere algún cambio, algún bugfix, alguna nueva funcionalidad en general; se requiere regresar desde el punto 0 de las etapas, para volver a pasar por desarrollo, por QA, y volver a llegar aquí.

Importante

Como desarrollador, debemos limpiar la falsa idea de que tenemos al equipo de QA como “contendientes” solo porque son quienes enuncian los defectos de nuestros aplicativos.

Al final, recuerda que cada defecto que arreglemos es un peldaño más hacia un aplicativo más sólido.

¡Siempre se trata de un ganar-ganar!

¿Qué es “pasar” el aplicativo?

Cuando decimos que un aplicativo “pasa” de un lado a otro, hacemos referencia a cada uno de los stages que hayan sido definidos por el equipo. Un stage representa un checkpoint en el aplicativo, y habitualmente toma raíz de una rama de GitHub.

Así, el desarrollador puede estar trabajando en la rama **develop1**, la cual habrá sido derivada de la rama **development**, esta rama development más adelante se mezclará con la rama **QA**, y finalmente QA podrá mezclarse con **main**, cuando éste se considere listo.

Recuerda que al final, la rama **main** siempre se considerará como la rama principal, y nuestro objetivo como desarrolladores es que, los cambios que nosotros estamos haciendo desde **develop1**, siempre lleguen a estar presentes en **main**. ¡Entonces habremos llegado al entorno “productivo”!

Pero, ¿Qué tiene que ver esto de las ramas de git con el despliegue que se menciona en clase? Para ello primero tendremos que recordar la esencia de un **despliegue**

¿De qué se trata un despliegue?

¿Recuerdas el despliegue del chat que hicimos con **glitch.com**, o bien el seteo de pods que contenía el cluster de contenedores con **docker + kubernetes**? fueron **despliegues del aplicativo**.

Al finalizar estos procesos, **conseguimos que la aplicación estuviera ejecutándose en un servidor diferente**, logrando así que estuviera activo siempre que el servidor mantuviera vivo el proceso.

El servidor contaba ya con un dominio o un link particular que podía ser visitado por cualquier persona, consiguiendo así que el cliente pueda visualizar los cambios como si estuviera visitando una página real.

CODERHOUSE

Herramientas de despliegue

El proceso de despliegue es **una de las partes más importantes dentro del flujo de vida de nuestro aplicativo**. Aquí decidiremos dónde terminará ejecutándose el producto final.

Hay múltiples alternativas para contar con un servidor en la nube. De manera que tenemos que elegir cuál puede sernos más útil acorde con nuestras necesidades.

Podemos optar por utilizar algún servicio robusto para aplicaciones de calibre enterprise (Azure, AWS, Google Cloud, etc.), o bien utilizar plataformas de calibre más simple.



CODERHOUSE

Alternativas gratuitas

No es común que un desarrollador dé pauta a invertir en una plataforma para hacer despliegues más complejos, cuando recién está comenzando a entender el “qué es”. Por ello, se recomienda tomar alguna alternativa gratuita para que puedas experimentar cuál es el flujo de despliegue de un aplicativo.

Una vez que tengas control de los procesos de despliegue elementales, será buen momento para poder comenzar a utilizar plataformas más robustas. ¡Vamos paso a paso! En este caso, el aplicativo a utilizar será **Railway.app** o **Render**



CODERHOUSE

Deploy de nuestro aplicativo con Railway.app

CODERHOUSE

Sobre Railway.app

Railway.app es una de las alternativas que podemos utilizar para realizar el despliegue de nuestra aplicación. Nos permitirá setear una app bastante sencilla para poder alojar nuestro código y ejecutarlo dentro.

Railway.app es de pago, sin embargo, nos brinda 5 USD cada mes para practicar nuestros despliegues. De manera que no tendremos de que preocuparnos.



Antes de comenzar, necesitamos un aplicativo para desplegar

Seguramente sabemos de qué estamos hablando: **Adoptme**. El proyecto sobre el cual hemos estado trabajando este tiempo.

La idea será configurar el aplicativo hasta que esté listo para ser desplegado, posteriormente, se subirá a Railway.app para tener un link que permita que nuestro backend pueda ser visitado donde sea que se ocupe.

Lo primero será crear una cuenta de Railway.app. Además, sobre el proyecto de Adoptme, esta vez se te recomienda que hagas un forkeo del proyecto, ya que necesitarás manejar las branches de este proyecto.

Recuerda que clonar un proyecto es tener la referencia de ese proyecto, **forkearlo** es generar una copia personal del mismo.



Checkpoint: Espacio de preparación

Toma un tiempo para preparar tu proyecto [Adoptme](#). Revisa el código e instala las dependencias indicadas. Además, recuerda crear tu cuenta en [Railway.app](#)

Tiempo estimado: **10 minutos**

 APROXIMACIÓN AL PROCESO

¡Comenzamos!

Con nuestro proyecto forkeado y nuestra cuenta de Railway.app creada, lo primero que haremos será comenzar a corroborar que los elementos fundamentales se encuentren en el aplicativo para poder ejecutarse arriba en Railway. Si no se cumplen estos criterios, se recomienda no comenzar nuestro deployment hasta que estén subsanados.

Los puntos a revisar son:

- ✓ **Variables de entorno seteadas**
- ✓ **Script de inicio**
- ✓ **Puerto de escucha**

APROXIMACIÓN AL PROCESO

Variables de entorno seteadas

Si inspeccionamos nuestro archivo app.js, notamos que están solicitándonos una URL de MONGO ATLAS, para poder conectar correctamente con la base de datos, sin embargo, sabemos que si subimos esta URL de manera limpia, Github se enojará por tener datos sensibles expuestos, por ello, colocaremos nuestra URL de la siguiente manera:

```
const connection = mongoose.connect(process.env.MONGO_URL)
```

También podríamos configurar nuestras variables de entorno del controlador de sessions (por toda la lógica de jwt y cookies). Sin embargo, de momento no es el objetivo principal, con la variable de entorno de la conexión a ATLAS estaremos bien.

Importante

¿Realmente necesitamos dotenv para nuestro proyecto?

Recuerda que, al desplegar nuestra aplicación en Railway, podremos colocar variables de entorno para dicho aplicativo, de manera que dotenv no será necesario para este ejemplo.

Si quisieramos ejecutar el proyecto sin dotenv, **recuerda que siempre podemos ejecutar en modo debug** nuestro aplicativo.

APROXIMACIÓN AL PROCESO

Script de inicio

¿Cuándo fue la última vez que corriste el script start en tu backend? Hemos estado tan acostumbrados a correr nuestro aplicativo a partir de **dev** (por el uso de nodemon), que en muchas ocasiones los desarrolladores llegan a olvidar colocar su script **start**

```
"scripts": {  
  "start": "node src/app.js",  
  "dev": "nodemon src/app.js",  
}
```

Este script **start** es crucial para el servidor una vez deployado, debido a que éste ejecuta este comando, además, es importante que el script inicialice el aplicativo con node y no con nodemon, ya que la dependencia nodemon es algo externo que nosotros instalamos global, pero que el servidor no conoce.

 APROXIMACIÓN AL PROCESO

Puerto de escucha

Por último, la forma en la que declaramos el puerto es importante también. Cuando ejecutamos el aplicativo desde nuestra computadora, abre el puerto que nosotros le indiquemos.

Contrario a esto, cuando nosotros desplegamos un aplicativo, el servidor es consciente de los puertos que tiene abiertos, por lo tanto, somos nosotros quienes deben acoplarse a este puerto, es por ello que la configuración de nuestro puerto debe verse así:

```
const PORT = process.env.PORT || 8080;
```

¿Nuestro aplicativo ya cumplió con estos puntos?
¡Vamos a Railway!

Recuerda hacer push de los cambios realizados

Ya que hicimos un cambio de la variable de entorno de mongo, es importante que hagamos push de nuestros últimos cambios a la rama principal.

Sabiendo que Railway tomará el repositorio de GitHub como referencia de despliegue, es importante siempre tener actualizado el repositorio.

```
git add .  
git commit -m "Added ENV VARIABLE For MONGO_URL"  
git push origin main
```

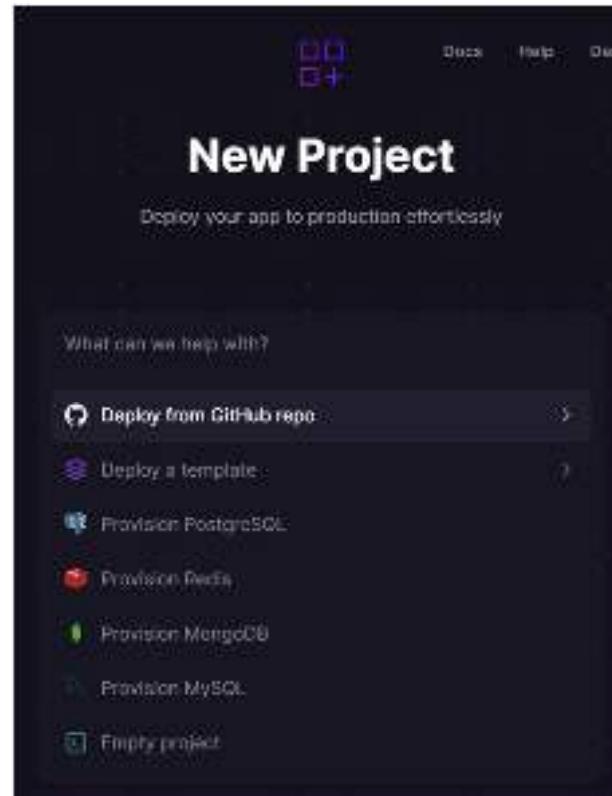
APROXIMACIÓN AL PROCESO

Creando un proyecto

Ahora, desde Railway seleccionaremos la opción “New Project”

Nota cómo nos da la opción directamente de hacerlo desde Github, ahí es donde toma acción nuestro proyecto recién forkeado.

Daremos click a la opción de Github y seleccionaremos el repositorio que recién forkeamos.

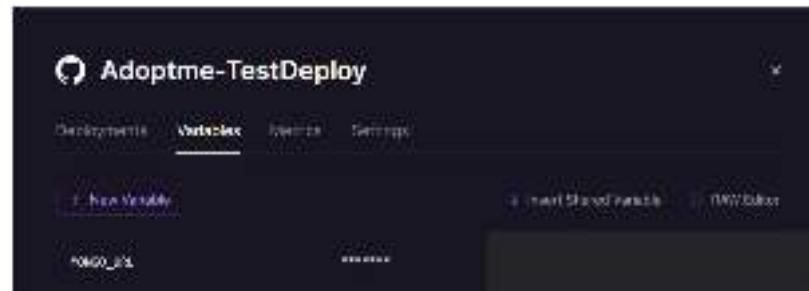
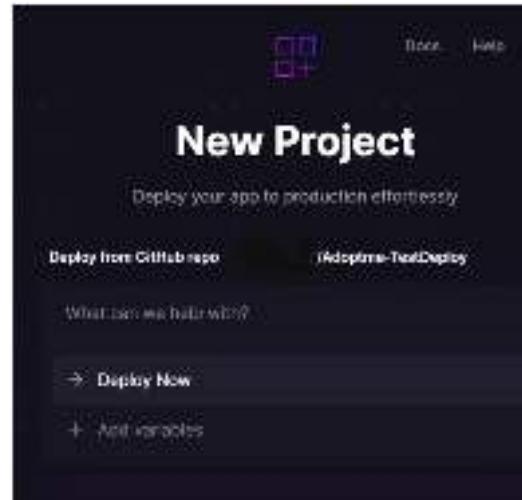


APROXIMACIÓN AL PROCESO

Configuración inicial

Antes de que se levante el aplicativo, Railway nos pregunta si tenemos intención de agregar variables de entorno, ¡Y nosotros tenemos una variable de entorno por agregar!

Daremos click en **Add variables**, y posteriormente añadiremos la MONGO_URL

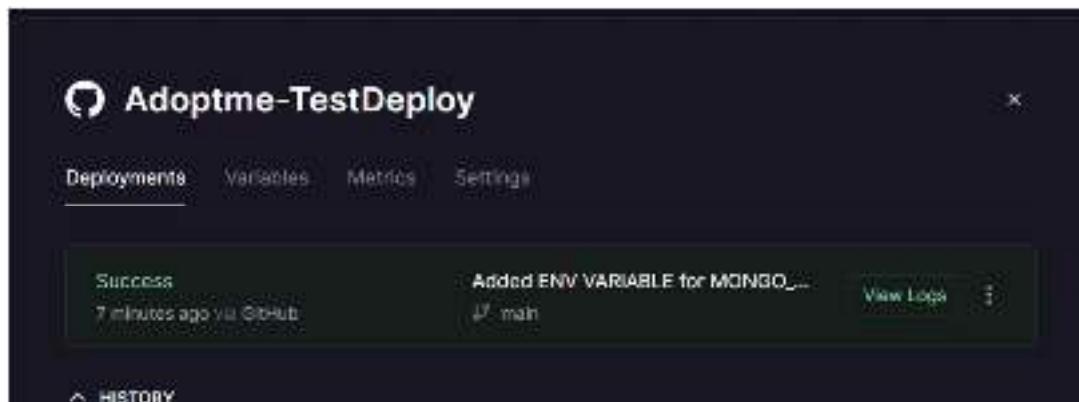


CODERHOUSE

APROXIMACIÓN AL PROCESO

Información del deploy

Podemos visualizar en un panel de información, en la pestaña de Deployments, el estatus actual del deploy que realizó la aplicación (Success en este caso). Siempre que un deploy salga mal, podemos dar click en **View Logs** para saber qué es lo que está saliendo mal

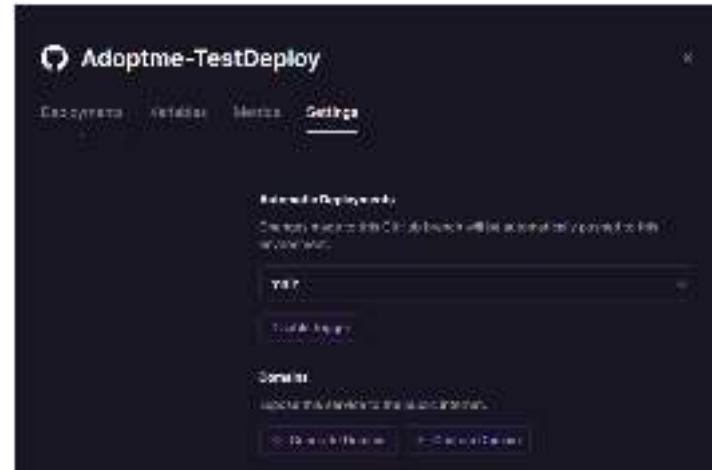


APROXIMACIÓN AL PROCESO

Generando un dominio

Por defecto, la aplicación se encuentra desplegada, pero no con un dominio especificado, para esto iremos a la pestaña de **Settings**. Luego, en el apartado **Domains**, seleccionaremos el botón **Generate Domain**

El link generado será el enlace para visitar nuestro backend.



APROXIMACIÓN AL PROCESO

Finalmente, utilizando nuestro backend

Si todos los pasos se siguieron correctamente, podemos visitar /api/users y obtendremos el resultado de usuarios de la base de datos de atlas que configuramos en nuestra MONGO_URL



The screenshot shows a terminal window with a curl command being run. The command is:

```
curl -X GET https://adoptme-testdeploy-production.up.railway.app/api/users
```

The output of the command is:

```
{"status": "success", "payload": [{"_id": "63a1d4264f84a5e72a071414", "first_name": "Mauricio", "last_name": "Espinosa", "email": "correo@correo.com", "role": "user", "pets": [], "__v": 0}]} 
```



Break

¡10 minutos y volvemos!

CODERHOUSE

Configuración de pipeline en Railway.app

CODERHOUSE

¿Qué es un pipeline?

Se entiende por pipeline a todo el flujo que comprende un proceso. Una característica importante de un pipeline es el uso de diferentes **stages**.

Estas stages están relacionadas con:

- ✓ **Diferentes entornos:** Cada entorno debe apuntar a diferentes bases de datos, por ejemplo, no es prudente mezclar mock users con usuarios reales de un entorno productivo.
- ✓ **Diferentes intenciones:** Una stage de desarrollo tiene la principal intención de corroborar que todas las ramas mergeadas de diferentes desarrolladores del equipo no hayan generado ningún conflicto funcional, sin embargo, una stage de QA no está interesada por la integración, sino directamente el carácter funcional del mismo.

Paso a paso

1

Desarrollo (preproductivo)

Probamos que todos los subdesarrollos de todos los devs estén correctamente integrados

2

QA (testing)

Corroboramos que los módulos no tengan defectos y mantengan una correcta funcionalidad.

3

Productivo

El aplicativo llega al cliente, está listo para poder ser utilizado en contextos reales.

Para trabajar con stages, también debemos pensar en branches

```
git checkout -b development  
git push origin development
```

```
git checkout -b QualityAssurance  
git push origin QualityAssurance
```

Hasta este momento solo hemos trabajado sobre la rama main del proyecto de Adoptme. La idea de haber forkeado el proyecto, también es que podamos separarlo en las branches que nosotros deseemos. Generaremos otras dos branches adicionales: development y QualityAssurance

Generaremos ambas ramas tanto en nuestro repositorio local, como en el repositorio remoto.

Configurando el pipeline en Railway.app

Al haber creado nuestra aplicación de Adoptme en Railway, esta se creó por defecto en un entorno de producción. Sin embargo, existe la forma de colocar múltiples entornos.

Daremos click en la opción “production” y seleccionaremos la opción **+ Environment**.



CODERHOUSE

Creación de entornos

Al dar click a la opción indicada, nos llevará a la configuración de environments, aquí es donde crearemos los dos environments diferentes: PreProduction y QualityAssurance

Ahora podremos visualizar los diferentes entornos disponibles.

¡Cuidado, entorno inestable!

Al crear un entorno, el mismo intentará desplegarse automáticamente. Sin embargo:

- ✓ Sigue en la misma stage que el entorno principal, por lo que hay que cambiarlo para que escuche por uno diferente.
- ✓ Un entorno vuelve a crearse vacío en sus variables de entorno, por lo que hay que reconfigurar nuestra variable MONGO_URL nuevamente, apuntando a una base distinta.



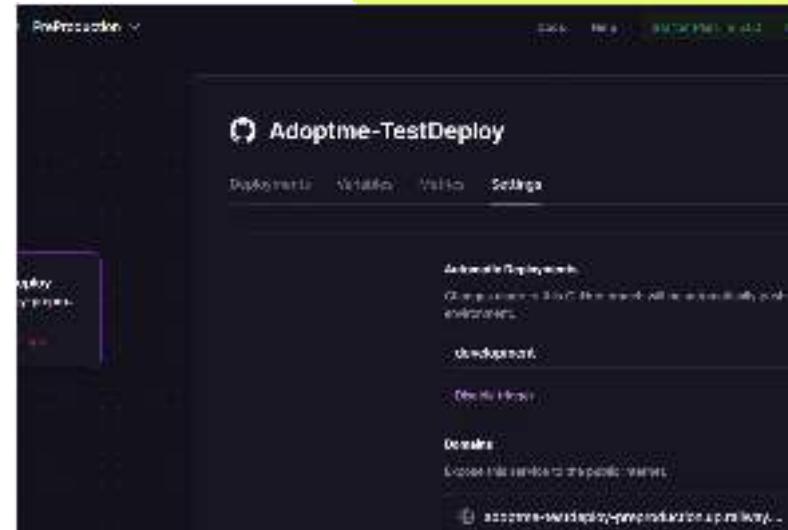
CODERHOUSE

¿Cómo reconfigurar los nuevos entornos?

Primero tenemos que ir a la configuración del respectivo entorno, y cambiaremos en **Automatic Deployments**, la rama por la cual queremos escuchar en dicho entorno. Colocaremos la rama development.

Además volveremos a colocar la URL de la base de datos en MONGO_URL, apuntando a una base dev, con el fin de que no afecte al entorno productivo.

Realizar el mismo proceso para el entorno QualityAssurance.



CODERHOUSE



Hands on lab

En esta instancia de la clase **repasaremos** algunos de los conceptos vistos en clase con una aplicación

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **20 minutos**

CODERHOUSE



HANDS ON LAB

Creación de una vista a partir de un proceso de desarrollo completo.

Se nos ha solicitado crear vistas para el proyecto de Adoptme, el cual debe contar con:

- ✓ Una vista de bienvenida a partir de handlebars en la ruta base
- ✓ Una vista que permita visualizar a todos los usuarios registrados hasta el momento.
- ✓ Una vista que permita visualizar a todas las mascotas registradas hasta el momento.



Creación de una vista a partir de un proceso de desarrollo completo.

Comenzar desde la etapa de desarrollo, contemplado que los cambios en desarrollo se visualicen solo en el entorno de desarrollo, mas no en QA ni en Productivo.

Posteriormente, realizar un Pull Request entre development y QualityAssurance con el fin de que ahora los cambios se visualicen en el nuevo link, sin embargo, se necesitarán llenar los campos nuevamente, debido a que se estará apuntando a una nueva base de datos.

Finalmente, realizar el mismo proceso de QA a Productivo y corroborar que la vista coincida en productivo, llenar nuevamente la base con usuarios válidos para el entorno productivo.
Al final, tendremos tres entornos, con diferentes datos, pero funcionales por sí solos



Dockerizando nuestro Proyecto

CODERHOUSE



ENTREGA DEL PROYECTO FINAL

Dockerizando nuestro Proyecto

Objetivos generales

- ✓ Implementar las últimas mejoras en nuestro proyecto y Dockerizarlo.

Objetivos específicos

- ✓ Documentar las rutas restantes de nuestro proyecto.
- ✓ Añadir los últimos tests
- ✓ Crear una imagen de Docker.

Se debe entregar

- ✓ Documentar con Swagger el módulo de "Users".

Se debe entregar

- ✓ Desarrollar los tests funcionales para todos los endpoints del router "adoption.router.js".
- ✓ Desarrollar el Dockerfile para generar una imagen del proyecto.
- ✓ Subir la imagen de Docker a Dockerhub y añadir en un ReadMe.md al proyecto que contenga el link de dicha imagen.



ENTREGA DEL PROYECTO FINAL

Dockerizando nuestro Proyecto

Formato

- ✓ Link al repositorio de Github con el proyecto (sin node_modules)
- ✓ Además, archivo .env para poder correr el proyecto.

Sugerencias

- ✓ Para repasar Docker, se recomienda revisar la clase 5 “Clusters & Escalabilidad”.

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Despliegue de aplicación
- ✓ Despliegue con Railway.app
- ✓ Pipeline de despliegue

CODERHOUSE

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser

- grabada

Clase 2. Testing y Escalabilidad Backend

Optimización

CODERHOUSE

Temario

1

Mocks

- ✓ TDD
- ✓ Desarrollo de prácticas preventivas a partir de mocks

2

Optimización

- ✓ Rendimiento en producción
- ✓ Comprensión con Brotli
- ✓ Middleware para manejo de errores

3

Versiones y paquetes

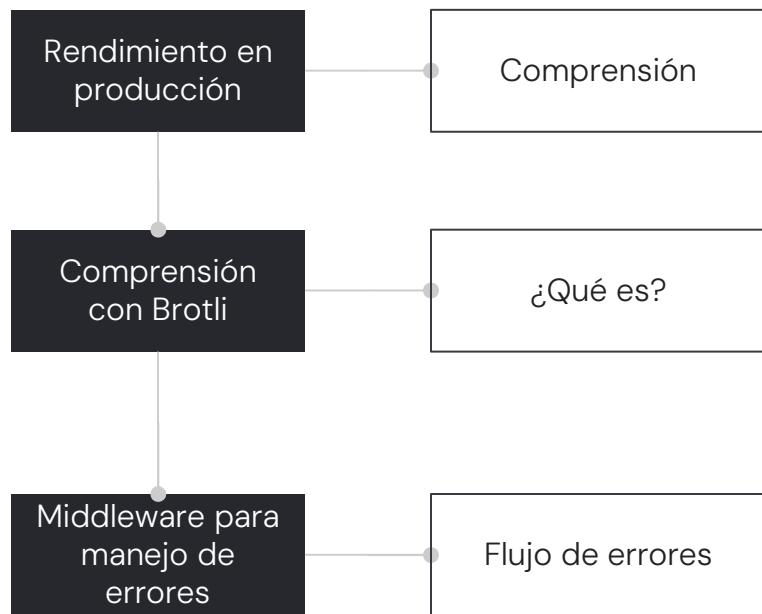
- ✓ Node Version Manager
- ✓ Administradores de paquetes
- ✓ Profundizando NPM

Objetivos de la clase

- Entender las implicaciones de un servidor en producción
- Teorizar la optimización general de un servidor
- Desarrollar un modelo práctico de optimización.



MAPA DE CONCEPTOS



Rendimiento en producción

CODERHOUSE

La realidad de un servidor

Cuando comenzamos a “acostumbrarnos” a desarrollar módulos y funcionalidades en nuestro servidor, seguramente comenzamos a agregar múltiples cosas mientras continuamos aprendiendo a utilizar el universo de herramientas para nuestro servidor.

Sin embargo, cuando queremos mostrar nuestro servidor al mundo, nos encontramos con una terrible realidad: la experiencia del usuario puede verse afectada debido a la saturación de cálculos, herramientas y funcionalidades por cargar.

¿Qué tanto de lo que colocamos en un servidor, es realmente útil para el usuario?



CODERHOUSE

¿Cómo optimizar el rendimiento del servidor?

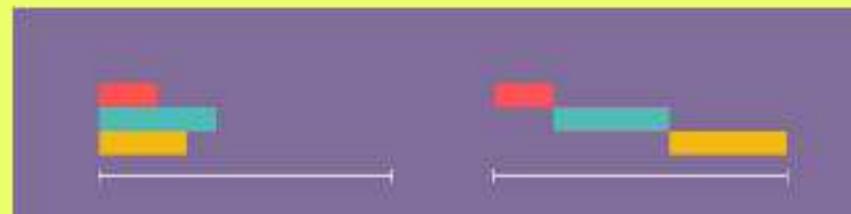
Para que el cliente pueda contar con una experiencia de usuario satisfactoria al utilizar nuestro sistema, podemos implementar algunas prácticas de optimización y hacer un flujo sólido y constante.

A continuación, se te brindarán un conjunto de prácticas que te ayudarán a asegurar un desarrollo de un servidor correctamente optimizado, sustentando una experiencia de usuario agradable.

1. Utilizar funciones asíncronas

Una de las principales ventajas de Javascript son las funciones asíncronas. Esto es debido al carácter no bloqueante con el que éstas cuentan en cuanto a sus operaciones.

Las funciones asíncronas permitirán que algunas otras tareas puedan ejecutarse mientras desarrollamos el trabajo principal, permitiendo hacer tareas como escribir archivos, leer datos de una base



Asynchronous vs Synchronous

2. Realiza un correcto loggeo

Aunque no lo pareciera, los `console.logs()` que vamos dejando a lo largo de nuestro flujo de desarrollo, pueden llegar a ser perjudiciales al hacer la medición del rendimiento de las tareas de nuestro servidor.

Un logger permitirá mostrar sólo lo necesario según el entorno en el que se corra el aplicativo, permitiéndonos loguear en consola cuando estemos desarrollando, pero ocultarlo cuando estemos en producción. Más adelante aprenderás cómo implementar un logger para tu servidor.



3. Usa una variable de entorno NODE_ENV = production

Express cuenta con una configuración interna que podemos aprovechar colocando esta simple variable en nuestro entorno.

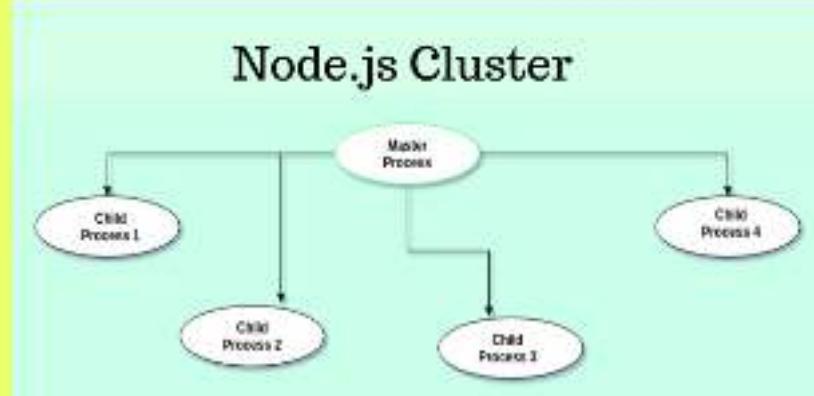
Setear NODE_ENV = production hará que Express de manera interna cambie algunas cosas en su configuración, como es:

- ✓ Guardar en caché templates de vistas
- ✓ Guardar en caché también archivos de estilos de extensiones CSS
- ✓ Generar menos mensajes “verbose” de errores. Esto quiere decir que no se ejecutarán los stack traces de los errores.

4. Clusterizar nuestra aplicación

Existe una forma de aprovechar los hilos o CPUs de un computador, para levantar múltiples instancias del servidor y poder atender un mayor número de peticiones concurrentes.

La clusterización nos ayudará a tener un “equipo de aplicaciones” para atender las tareas más complicadas.



5. Configurar el servidor para que éste se reinicie automáticamente.

En caso de que un error llegara a ocurrir a lo largo de la aplicación, nuestro servidor podría verse afectado al punto de que éste se caiga.

Sabemos que para estos casos necesitamos reiniciar el servidor de manera manual, pero claramente no es lo más factible

Debe existir algo que permita mantener el servidor con vida, aún ante errores fatales.

Si implementamos lo anterior, podremos mantener múltiples instancias de nuestra aplicación, aunque también podemos utilizar un **manejador de procesos**.

Un manejador de procesos nos permitirá escuchar por estas instancias de aplicación y reiniciar alguna de éstas en caso de que algo malo llegara a suceder con alguna.



6. Manejar correctamente errores

Recordemos que un error puede suponer algo fatal para el flujo de una aplicación. Ciertos errores pueden hacer inútil el continuar con los procesos y pueden dejar el servidor abajo mientras alguien lo levanta nuevamente.

Por ello necesitamos asegurar que nuestro código esté lo mejor controlado posible en cuanto a errores, asegurando que, en caso de ocurrir algo no esperado, siempre podamos impedir que el servidor se venga abajo.

Si implementamos lo indicado en el número 5, nuestro servidor se reiniciará, pero generará tiempos de reinicio innecesario, idealmente esperamos que **nunca** se caiga el servidor.

Para ello, podemos utilizar try-catchs, con promesas, e implementar algunos error handlers para impedir que lleguen a puntos críticos del servidor.

7. Realizar balanceos de carga

Algunos servidores o algunas instancias pueden procesar más recursos que otros, por ello, cuando múltiples servidores trabajan en conjunto, debemos poder contar con la posibilidad de decidir a qué servidor le debería corresponder un mayor número de peticiones.

De esta manera, el servidor A puede soportar tres veces más que el servidor B y C, o bien podemos hacer que el servidor A se encargue de soportar las peticiones que correspondan a procesos más complejos, mientras que B y C pueden encargarse de las peticiones más simples.

8. Realizar compresión

La compresión es un recurso utilizado para poder realizar un número más reducido de transferencia, aún manteniendo el mismo contenido de data.

La compresión permitirá que nuestros archivos puedan viajar a través de la red de manera más rápida, manteniendo la consistencia.

Hay que tener en cuenta que realizar una compresión de datos puede resultar en un procesamiento algo complejo dejándose directamente a cargo del servidor (en algún middleware, por ejemplo)

Existe una compensación en compresión/procesamiento, sin embargo, debemos tomar en cuenta estas posibles implementaciones para saber dónde colocarlo correctamente.

9. Utilizar un proxy inverso

Utilizar el punto intermedio entre el servidor y el cliente, como un proxy inverso, nos permitirá tener una mejor gestión de los servidores con los cuales estamos trabajando para el desarrollo de nuestra aplicación.

Un proxy inverso nos será útil para muchas cosas

Algunas de las ventajas que podemos obtener de un proxy inverso son:

- ✓ Anonimato de servidores, generando un punto de identificación general y redireccionando a los servidores.
- ✓ Realizar carga de archivos estáticos, para reducir la carga de trabajo al servidor
- ✓ Realizar balances de carga y compresiones.

¡Importante!

¡Aún hay mucho por aprender! Una vez que nuestro servidor ha cumplido con su función principal, es momento de comenzar a pensar en optimizarlo, mantenerlo y prepararlo para entornos empresariales reales.

Compresión

CODERHOUSE

¿Qué es la compresión?



Cuando el servidor se está comunicando con el cliente en el navegador, parte de esta comunicación implica revisar si hay algún archivo comprimido que necesite descomprimirse, y en caso de que así sea, cuál sería el algoritmo de descompresión a utilizar para poder obtener la información correctamente.

Los navegadores modernos pueden aceptar contenido codificado en tres algoritmos principales

- ✓ Deflate
- ✓ Gzip (Deflate + algunas cosas adicionales)
- ✓ Brotli

El día de hoy hablaremos sobre Gzip y Brotli

Gzip

Gzip es el primer y más conocido modelo de compresión, es altamente utilizado y sencillo de utilizar.

La compresión se puede colocar a nivel middleware en nuestro servidor para poder corroborar la diferencia de transferencia entre una respuesta con y sin compresión.

Para poder utilizarlo, instalaremos el módulo compression

```
npm install express-compression
```



CODERHOUSE



Ejemplo en vivo

Se levantará un servidor de express con un endpoint que responda con una string ridículamente grande.

Luego, se ejecutará el middleware de compresión para visualizar la diferencia.

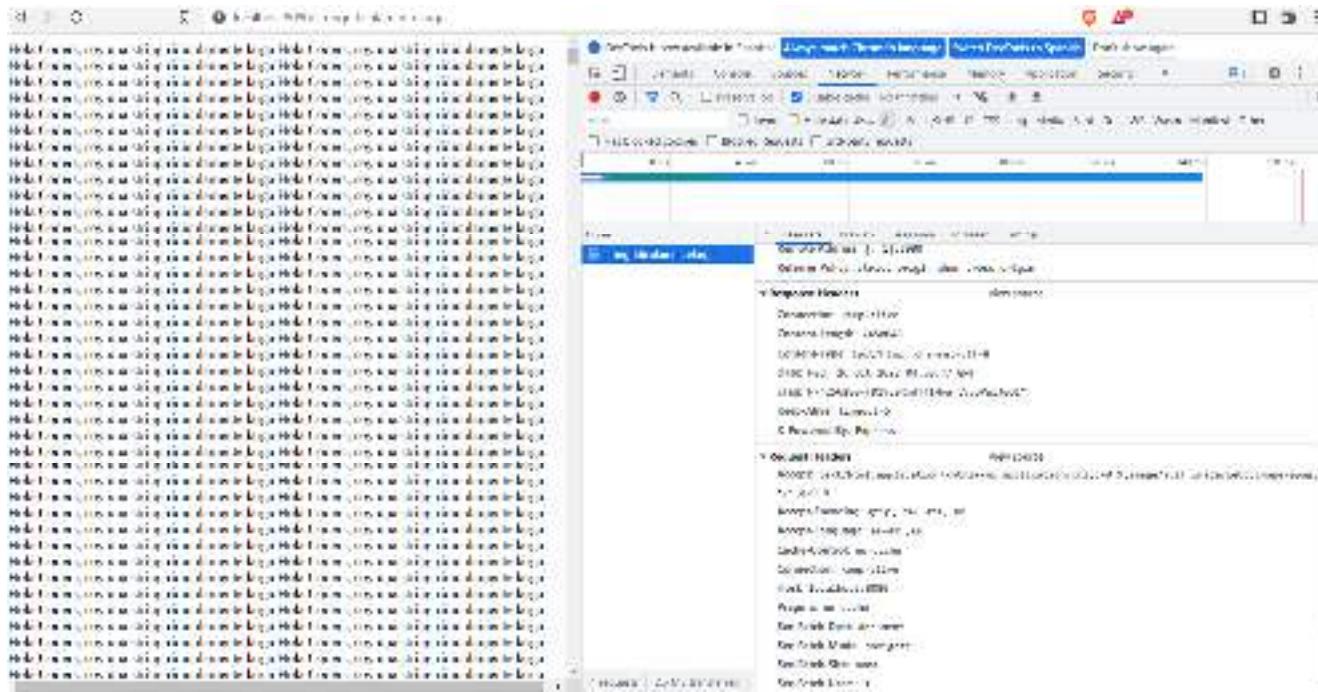
Duración: 15 min

CODERHOUSE

Ejemplo: endpoint

```
JS app.js    X  () package.json 1
src > JS app.js > ...
1 import express from 'express';
2
3 const app = express();
4
5 const server = app.listen(8080, ()=>console.log("Listening on 8080"));
6
7 app.get('/stringridiculamentelarga',(req,res)=>{
8     let string = `Hola Coders, soy una string ridiculamente larga`;
9     for(let i=0;i<10e10;i++){
10         string+=` Hola Coders, soy una string ridiculamente larga`;
11     }
12     res.send(string);
13 })
```

Ejemplo: prueba de endpoint



CODERHOUSE

Ejemplo: analizando info

The screenshot shows a file browser window with a file named "Caché deshabilitado para probar correctamente" selected. The file is a compressed archive with a size of 1.00 MB and a modification date of 2014-09-17 10:51:26. The browser interface includes tabs for Home, Folders, Files, and Recent, with a toolbar above.

Data transferida

Caché deshabilitado para probar correctamente

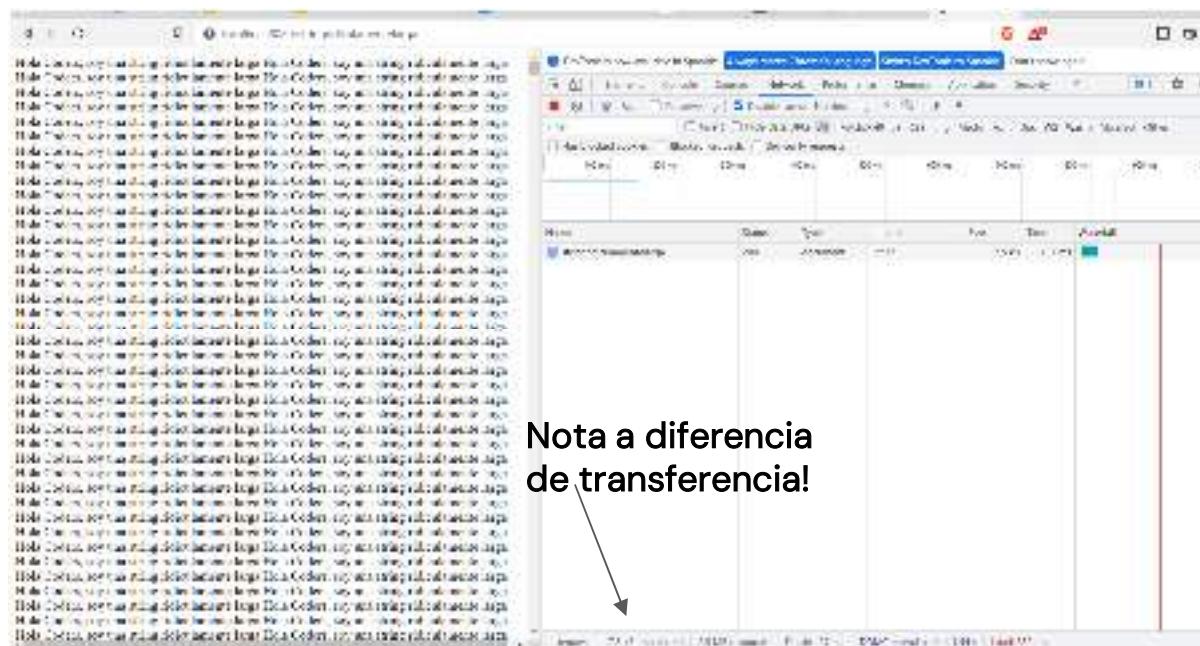
Qué algoritmos de compresión aceptaría

CODERHOUSE

Ejemplo: ejecutando middleware de compression

```
5 app.js  *  () package.json
src > 6 app.js > ...
1 import express from 'express';
2 import compression from 'express-compression';
3
4 const app = express();
5
6 const server = app.listen(8080, ()=>console.log("Listening on 8080"));
7
8
9 app.use(compression());
10 app.get('/stringridiculamentelarga',(req,res)=>{
11   let string = 'Hola Coders, soy una string rididulamente larga';
12   for(let i=0;i<5e4;i++){
13     string+=` Hola Coders, soy una string rididulamente larga`;
14   }
15   res.send(string);
16 })
```

Revisando nuevamente el resultado



Nota a diferencia
de transferencia!

Revisando nuevamente el resultado

The screenshot shows a browser developer tools Network tab with a single entry for the URL `http://localhost:8080/stringridiculamente larga`. The Request Headers section shows the client's Accept-Encoding header set to `gzip, deflate, br`. The Response Headers section shows the server's Content-Encoding header set to `gzip`, indicating that the response was compressed.

Indica que se aceptó el gzip configurado



Break

¡10 minutos y volvemos!

CODERHOUSE

Compresión con Brotli

CODERHOUSE

¿Qué es Brotli?

Brotli es conocido como una alternativa “moderna” de Gzip.

Éste fue desarrollado por Google y ofrece un algoritmo cuya compresión puede resultar hasta 30% más efectiva que la compresión de Gzip

Para ello, podemos seguir utilizando la dependencia de express-compression que recién instalamos, añadiendo un poco de configuración extra.

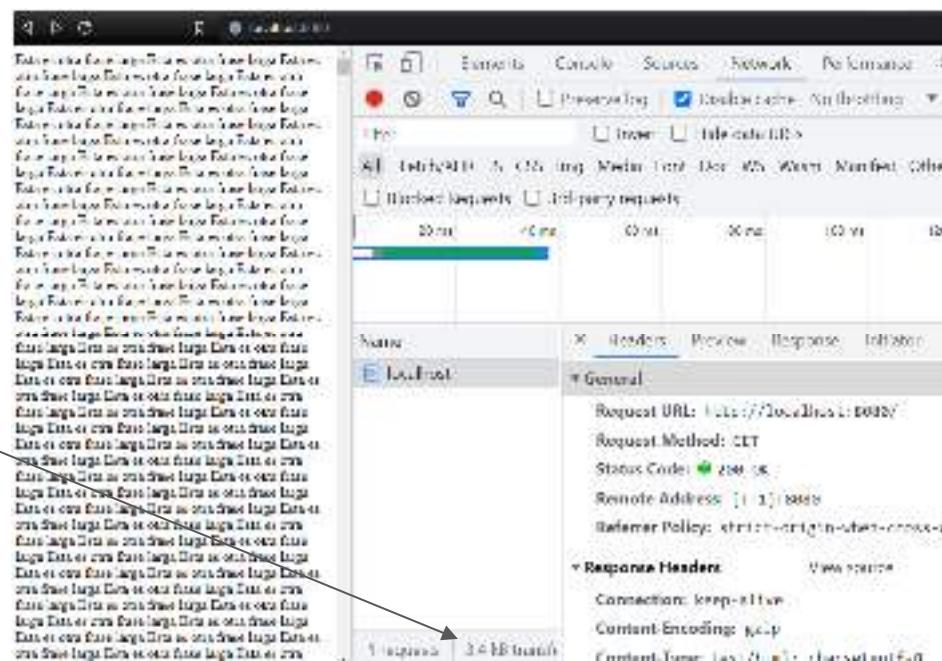
Implementación con compression de gzip

```
JS app.js    X  {} package.json

src > JS app.js > ...
  1 import express from 'express';
  2 import compression from 'express-compression';
  3 const app = express();
  4 const server = app.listen(8080, ()=>console.log("Listening"));
  5
  6 app.use(compression());
  7 app.get('/', (req, res)=>{
  8   let string = "Esta es otra frase larga";
  9   for(let i=0;i<5e4;i++){
10     string+=" Esta es otra frase larga";
11   }
12   res.send(string);
13 })
```

Nivel de compresión con gzip en endpoint

Compresión a
3.4kb



Realizando configuración para brotli

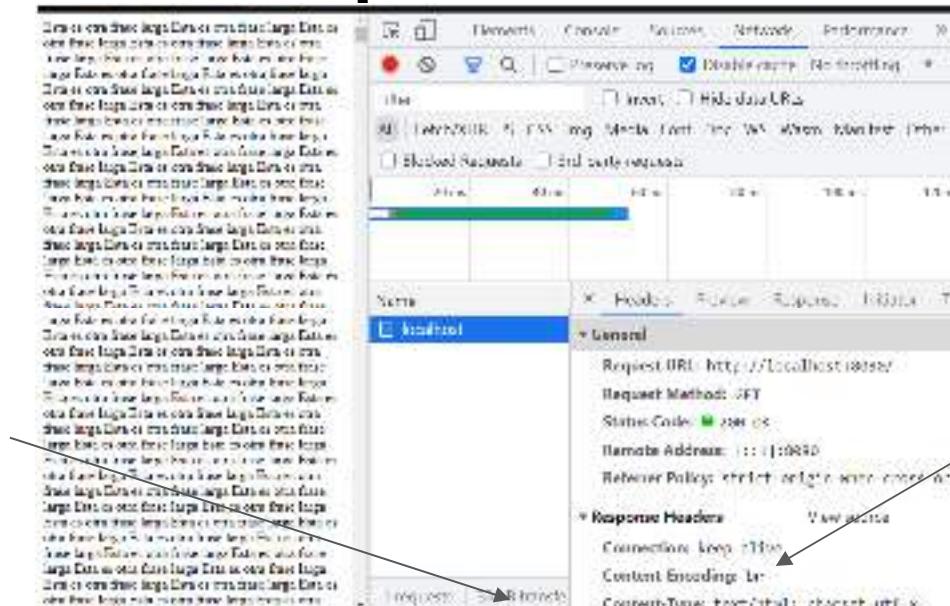
A nuestra configuración de compression, añadiremos los siguientes cambios dentro de la instancia del middleware.

```
app.use(compression({
  brotli:{enabled:true, zlib:{}}
}));
```

Con esta opción, se reconocerá un tipo de compresión “br” (brotli) al momento de enviar la información. La razón por la que colocamos un objeto zlib vacío se debe a que el módulo de express-compression cuenta con una dependencia interna “zlib”, la cual le permite ejecutar diferentes niveles de compresión.

Nivel de compresión con brotli en el mismo endpoint

Compresión a
335B. ¡Una clara
diferencia!



Nota cómo ha
cambiado el
content
encoding

¡Importante!

La compresión de brotli es considerablemente más efectiva que gzip. Sin embargo, utilizarlo como middleware implica un uso del algoritmo de compresión que consume más recursos, por lo que existe una nivelación entre tiempo de compresión y efectividad.

Middleware para manejo de errores

CODERHOUSE

¡Siempre atento a los fallos!

Aún con amplios años de experiencia, la probabilidad de cometer un error a lo largo de tu carrera como desarrollador **nunca será cero**.

Es por ello que, parte de tener un servidor correctamente optimizado, es también tener un correcto control de los errores que pueden llegar a ocurrir en el sistema.



CODERHOUSE

Flujo de errores

La idea será contar con nuestra propia gestión interna de errores, con el fin de mantener un mejor orden en el equipo de desarrollo.

Contar con una librería de errores permitirá que todos los desarrolladores puedan tener una referencia de los errores más comunes dentro de su aplicativo, apoyando así a su respectiva depuración

Para poder generar este flujo de error, necesitaremos de tres cosas primordiales:

- ✓ Un middleware de recepción de errores.
- ✓ Un generador personalizado de errores.
- ✓ Un diccionario de errores.



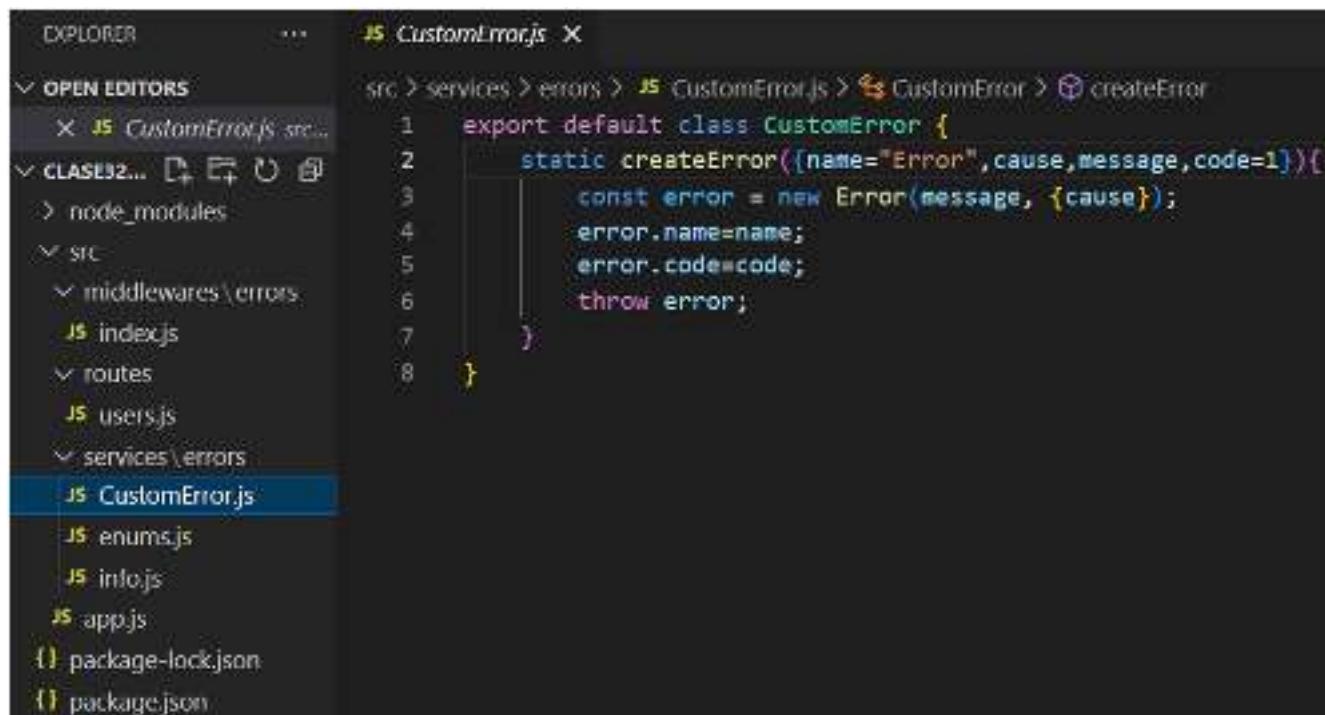
Ejemplo en vivo

Se realizará una implementación completa de manejo de errores personalizados, con el fin de mantener una correcta organización del servidor para el control de éstos.

Duración: 25 minutos

CODERHOUSE

Ejemplo: CustomError.js



The screenshot shows the VS Code interface with the Explorer and Editor panes visible.

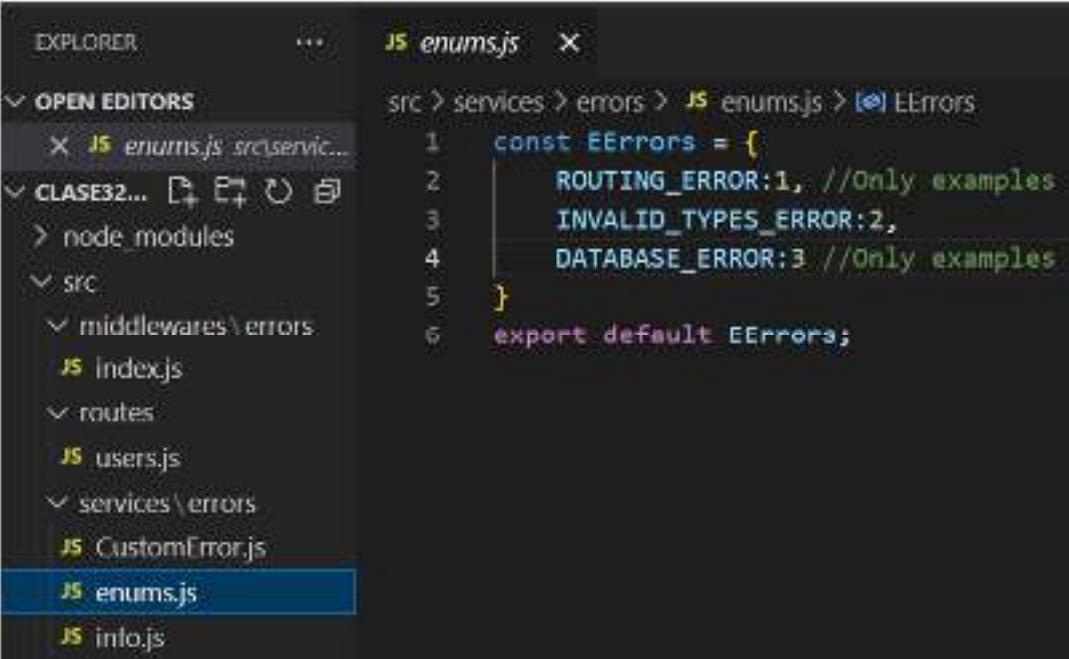
EXPLORER pane:

- OPEN EDITORS:
 - CustomError.js (selected)
 - index.js
 - users.js
 - services\errors\CustomError.js (highlighted)
 - enum.js
 - info.js
 - app.js
- CLASES... (button)
- node_modules
- SRC
 - middlewares\errors
 - index.js
 - routes
 - users.js
 - services\errors
 - CustomError.js (highlighted)

CustomError.js

```
src > services > errors > CustomError.js > CustomError > createError
1  export default class CustomError {
2      static createError([name="Error",cause,message,code=1]){
3          const error = new Error(message, {cause});
4          error.name=name;
5          error.code=code;
6          throw error;
7      }
8  }
```

Ejemplo: EErrors.js



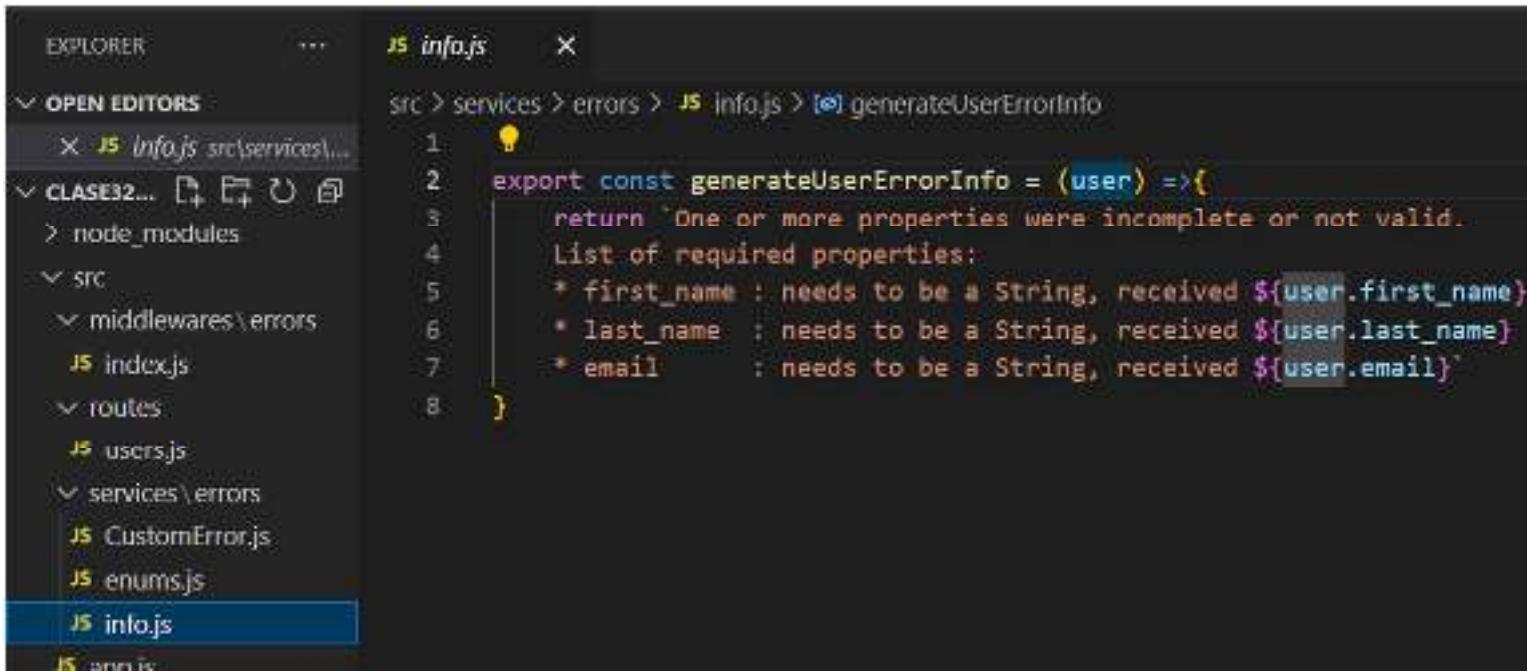
The screenshot shows a code editor interface with the following details:

- EXPLORER** sidebar:
 - OPEN EDITORS: `enums.js`
 - CLASE32... (with icons)
 - `node_modules`
 - SRC
 - `middlewares\errors`: `index.js`
 - `routes`: `users.js`
 - `services\errors`: `CustomError.js`
 - `enums.js` (highlighted with a blue selection bar)
 - `info.js`
- JS enums.js** editor tab:

src > services > errors > `enums.js` > `EErrors`

```
1 const EErrors = [
2   ROUTING_ERROR:1, //Only examples
3   INVALID_TYPES_ERROR:2,
4   DATABASE_ERROR:3 //Only examples
5 ]
6 export default EErrors;
```

Ejemplo: info.js



```
1  export const generateUserErrorInfo = (user) => {
2    return `One or more properties were incomplete or not valid.
3           List of required properties:
4           * first_name : needs to be a String, received ${user.first_name}
5           * last_name  : needs to be a String, received ${user.last_name}
6           * email       : needs to be a String, received ${user.email}`
7
8 }
```

Ejemplo: users.js (Parte I)

The screenshot shows the VS Code interface with the Explorer sidebar on the left and the code editor on the right.

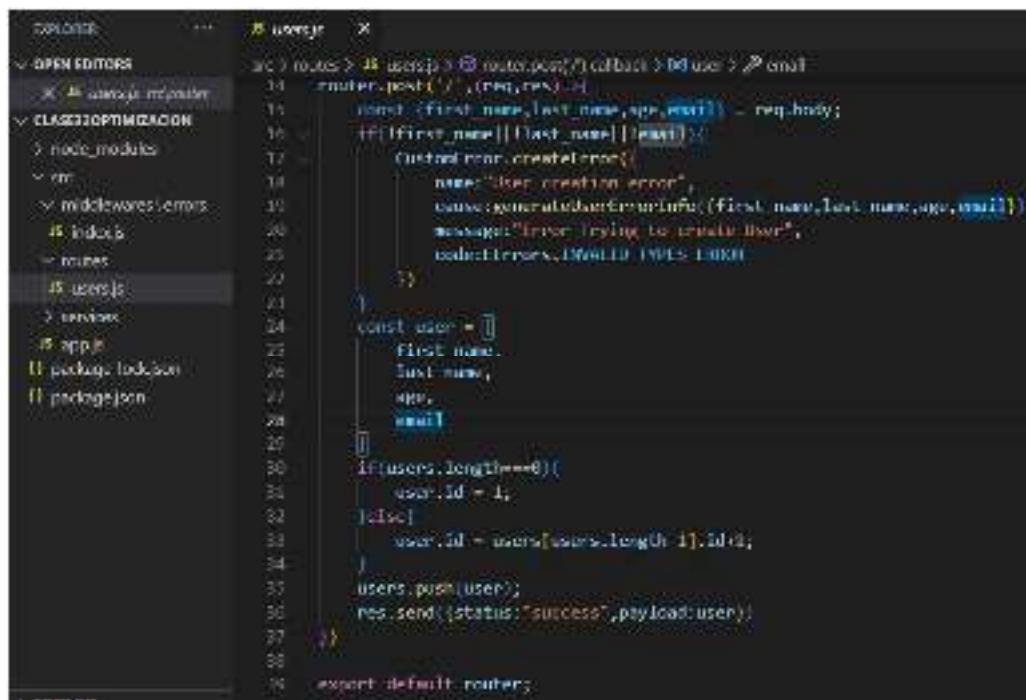
EXPLORER

- OPEN EDITORS
 - JS users.js** (selected)
 - JS users.js src\routes
 - CLASE32... [+]
 - > node_modules
 - src
 - middlewares\errors
 - JS index.js
 - routes
 - JS users.js** (selected)
 - > services\errors
 - JS app.js
 - { package-lock.json

JS users.js

```
src > routes > JS users.js > router.post('/').callback > code
1 import { Router } from 'express';
2 import CustomError from '../services/errors/CustomError.js';
3 import EErrors from '../services/errors/enums.js';
4 import { generateUserErrorInfo } from '../services/errors/info.js';
5
6 const users = [];
7
8 const router = Router();
9
10 router.get('/', (req, res) => {
11   res.send({ status: "success", payload: users })
12 }
13 )
```

Ejemplo: users.js (Parte II)

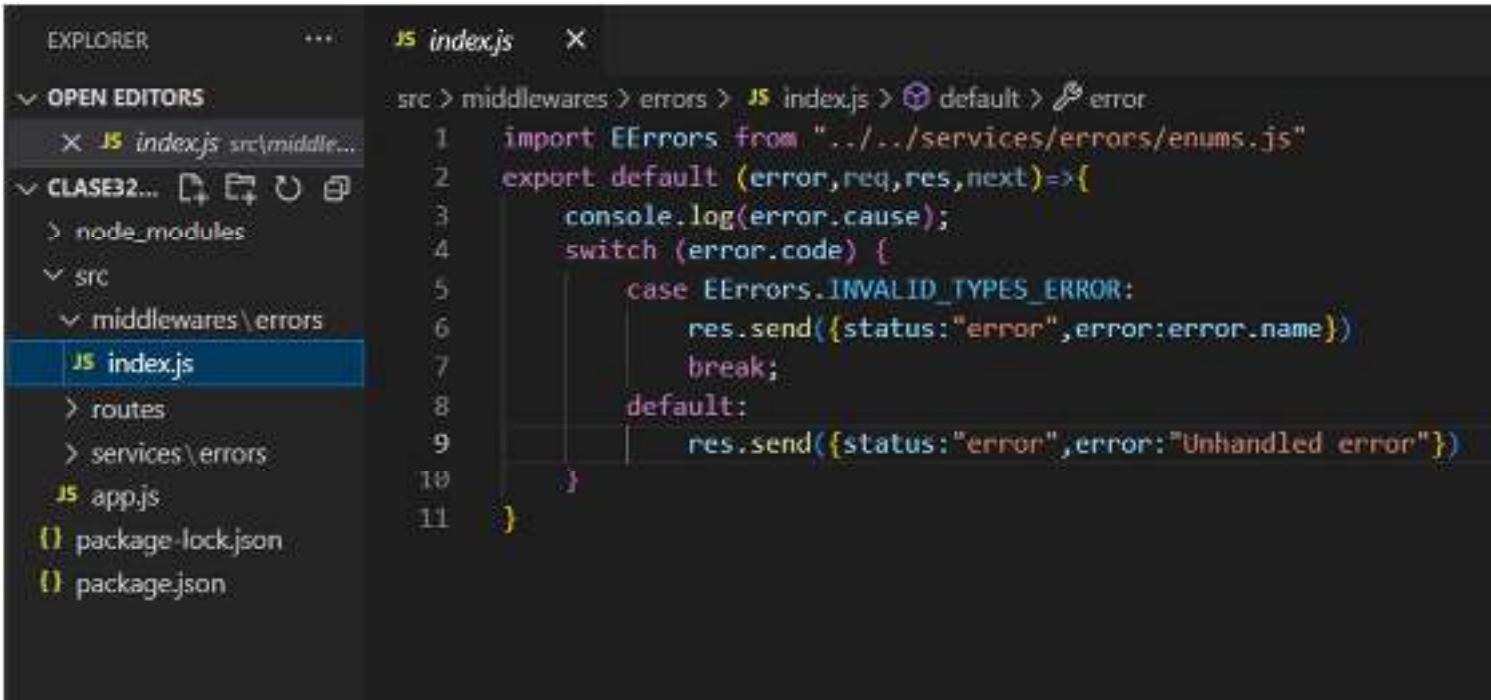


```
CODEER
OPEN EDITORS
CLASES2OPFMZACON
node_modules
cm
middlewares
  JS index
  routes
    JS users.js
  services
  app.js
  package-lock.json
  package.json

users.js

arc > routes > JS users.js @ routes.post() callback > 0d user > JS mail
  router.post('/users', async (req, res) =>
    const {first_name, last_name, age, email} = req.body;
    if(first_name || last_name || email) {
      const user = await User.create({
        name: "User creation error",
        user: generateRandomString(first_name, last_name, age, email),
        message: "Error trying to create user",
        collectionUsers: false
      });
      const user = []
      user.id = 1;
      user.first_name = first_name;
      user.last_name = last_name;
      user.age = age;
      user.email = email;
      if(users.length === 0){
        user.id = 1;
        user.id = users[users.length - 1].id + 1;
      }
      users.push(user);
      res.send({status: "success", payload: user});
    }
  );
  export default router;
```

Ejemplo: Error middleware.js



The screenshot shows the VS Code interface with the Explorer sidebar on the left and the code editor on the right.

EXPLORER sidebar:

- OPEN EDITORS:
 - index.js
- CLASE32...
 - node_modules
- src
 - middlewares\errors
 - index.js
 - routes
 - services\errors
 - app.js
- package-lock.json
- package.json

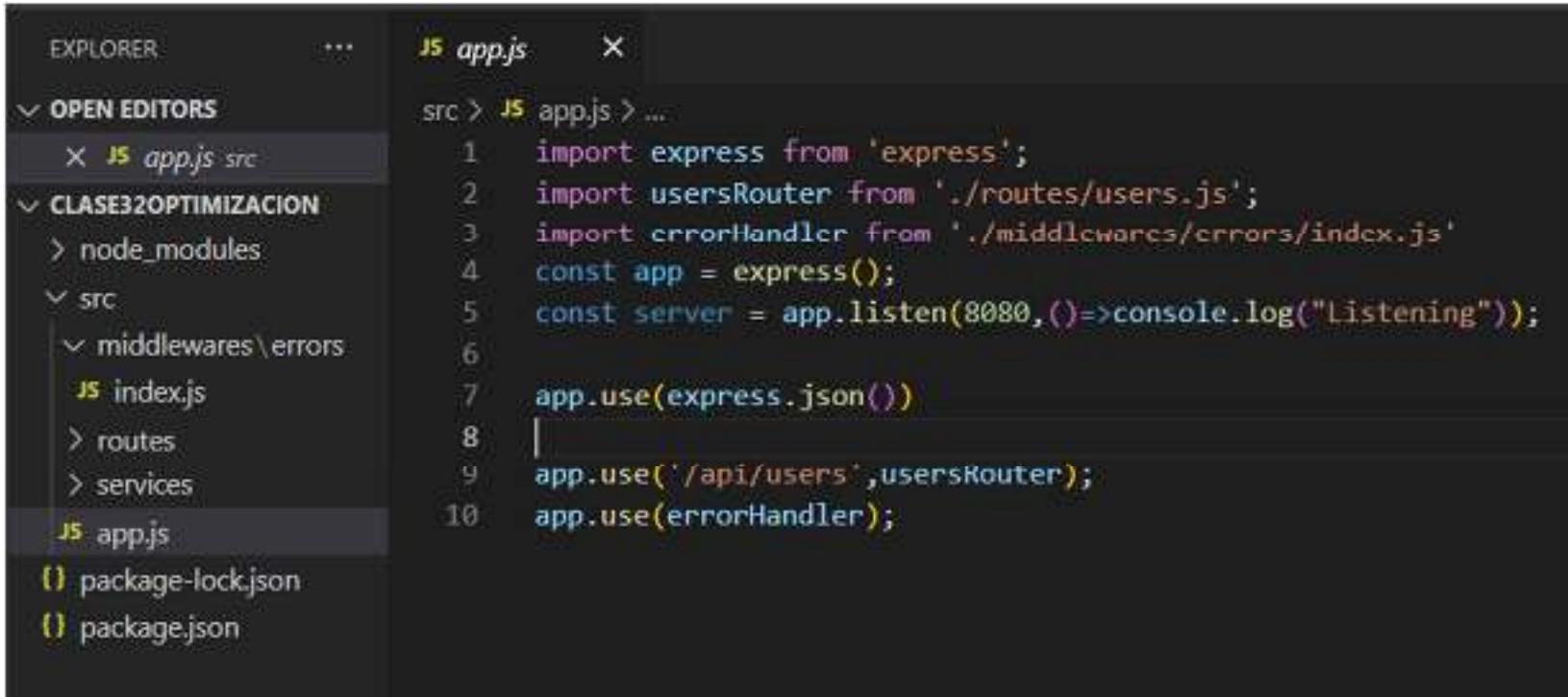
CODE EDITOR content (index.js):

```
index.js  X

src > middlewares > errors > index.js > default > error

1 import EErrors from "../../services/errors/enums.js"
2 export default (error,req,res,next)=>{
3     console.log(error.cause);
4     switch (error.code) {
5         case EErrors.INVALID_TYPES_ERROR:
6             res.send({status:"error",error:error.name})
7             break;
8         default:
9             res.send({status:"error",error:"Unhandled error"})
10    }
11 }
```

Ejemplo: app.js



The screenshot shows the VS Code interface with the Explorer sidebar on the left and the code editor on the right.

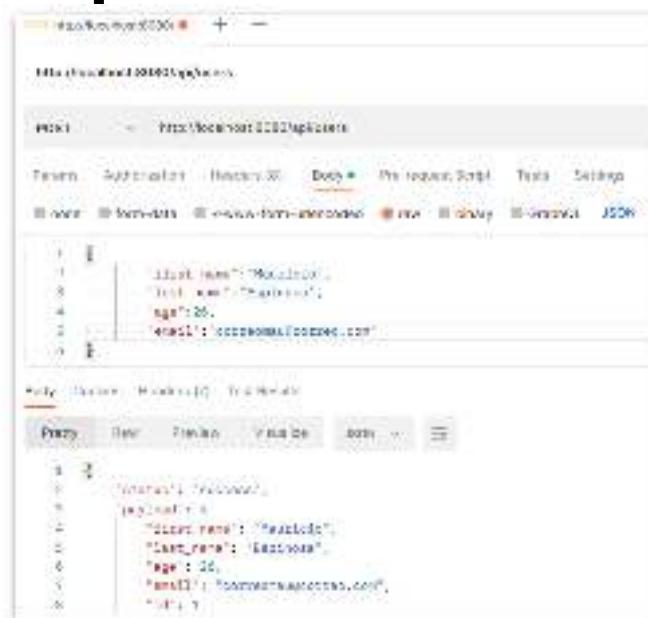
EXPLORER

- OPEN EDITORS
 - JS app.js src (highlighted)
- CLASE32OPTIMIZACION
 - node_modules
- src
 - middlewares\errors
 - JS index.js
 - routes
 - services
 - JS app.js (highlighted)
- package-lock.json
- package.json

app.js

```
src > JS app.js > ...
1 import express from 'express';
2 import usersRouter from './routes/users.js';
3 import errorHandler from './middlewares/errors/index.js';
4 const app = express();
5 const server = app.listen(8080, ()=>console.log("Listening"));
6
7 app.use(express.json())
8 |
9 app.use('/api/users',usersRouter);
10 app.use(errorHandler);
```

Ejemplo: Creando un usuario (Datos completos)



The screenshot shows a Postman interface with the following details:

- Request URL:** http://localhost:8080/api/users
- Method:** POST
- Body (JSON):**

```
1   "first_name": "Matilde",
2   "last_name": "Chaparro",
3   "age": 26,
4   "email": "correo@mail.compeo.com"
```

- Response (JSON):**

```
1   "id": 1,
2   "first_name": "Matilde",
3   "last_name": "Chaparro",
4   "age": 26,
5   "email": "correo@mail.compeo.com",
6   "status": 1
```

Ejemplo: Creando un usuario (Datos incompletos, generando el error)



Nota lo fácil que es reconocer errores particulares con nuestros propios logs de errores. ¡Depuraciones más efectivas!

```
[nodemon] starting `node src/app.js`
Listening
One or more properties were incomplete or not valid.
  List of required properties:
    * first_name : needs to be a String, received Mauricio
    * last_name  : needs to be a String, received Espinosa
    * email       : needs to be a String, received undefined
```



Manejador personalizado de errores

Duración: 5-10min

CODERHOUSE



ACTIVIDAD EN CLASE

Manejador personalizado de errores

Con base en el ejemplo de errores planteado.

Crear un endpoint en el router de usuarios router.get('/:uid') para recibir a un usuario. NO centrarse en la lógica para devolver al usuario, sino en el error en caso de que no envíen un parámetro numérico válido.

Complementar el código para que se pueda arrojar un error de tipo "INVALID_PARAM", en caso de que se quiera buscar a un usuario por un :uid inválido (por ejemplo, un valor no numérico, numérico negativo o undefined).

Gestionar el tipo de error en el Enum, en el middleware y en la info.



Mocking y manejo de errores



ACTIVIDAD PRÁCTICA

Mocking y manejo de errores

Consigna

- ✓ Se aplicará un módulo de mocking y un manejador de errores al proyecto [Adoptme](#).

Formato

- ✓ Link al repositorio de github sin node_modules

Sugerencias

- ✓ Céntrate solo en los errores más comunes

Aspectos a incluir

- ✓ Crear un módulo de Mocking para el servidor, con el fin de generar mascotas (sin **owner** y con **adopted** en "false") de acuerdo a un parámetro numérico. Utilizar este módulo en un endpoint GET llamado "/mockingpets" y generar 100 mascotas con el mismo formato que entregaría una petición de Mongo.
- ✓ Además, generar un customizador de errores y crear un diccionario para tus errores más comunes al registrar un usuario, crear una mascota, etc.

¿Preguntas?

CODERHOUSE

Resumen de la clase hoy

- ✓ Qué es el TDD
- ✓ Qué son los mocks
- ✓ faker js
- ✓ Aplicar los conceptos en una mocking API sencilla.

Opina y valora
esta clase

CODERHOUSE

Muchas gracias.

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser

- grabada

CODERHOUSE

Clase 4. Testing y Escalabilidad Backend

Logging y performance

CODERHOUSE

Temario

3

Versiones y paquetes

- ✓ Node Version Manager
- ✓ Administradores de paquetes
- ✓ Profundizando NPM

4

Logging y performance

- ✓ Loggers
- ✓ Testing de performance
- ✓ Testing avanzado de performance

5

Clusters y escalabilidad

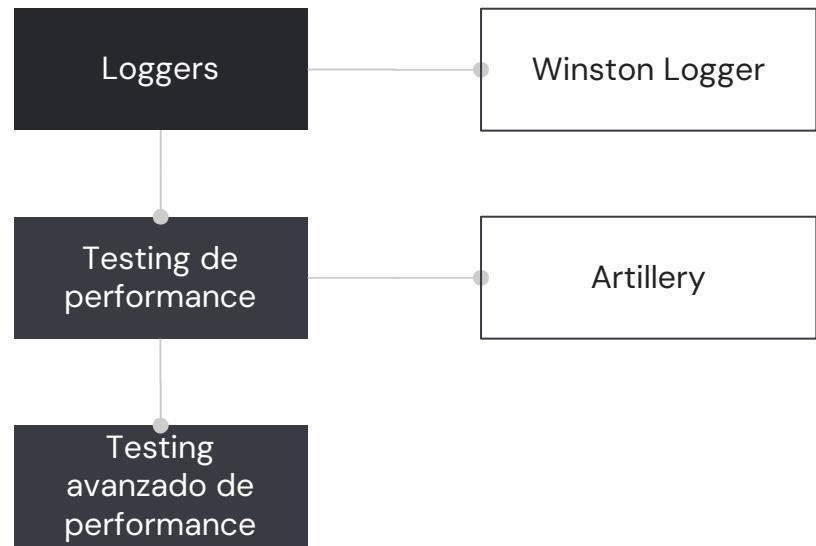
- ✓ Módulo Cluster
- ✓ Docker
- ✓ Docker como PM

Objetivos de la clase

- Conocer el logger de Winston.
- Conocer el test de carga de Artillery
- Realizar un modelo de performance de nivel avanzado con Artillery



MAPA DE CONCEPTOS



Loggers

CODERHOUSE

La consola como barrera en performance

Seguimos hablando de la importancia de que nuestro servidor sea eficiente al momento de salir a un entorno productivo. Esta vez nos toca abordar el problema que supone la típica línea **console.log()** que utilizamos desmedidamente.

¿Qué implicaciones pueden tener los console.logs y por qué hay que comenzar a evitarlos?

```
console.log("Hola mundo")
```

CODERHOUSE

Problemas con mostrar en consola

Lo primero es entender que es una acción síncrona, y que hay que utilizarla con cuidado, ya que un conjunto de console.logs agrupados en una función pueden generar un ligero estancamiento en la ejecución de la función.

Lo segundo es la persistencia del mismo. Si bien algunos logs no es importante conservarlos, hay algunos otros de mayor prioridad (warnings, errors, por ejemplo), que terminan perdiéndose entre una cantidad innecesaria de mensajes.

Solución: Loggers

Un logger nos permitirá mostrar información sobre nuestra aplicación, con algunas particularidades:

- Podemos mostrar las cosas a partir de “niveles” para separarlos por prioridad
- Podemos enviar información a otros recursos, no sólo a la consola, a partir de algo conocido como “transportes”

Además, gracias a lo anteriormente mencionado, podemos loggear diferentes niveles en diferentes transportes según el entorno, de manera que podemos limitar **qué es lo que llega a productivo** y qué se queda en desarrollo.

Winston Logger

CODERHOUSE

Winston Logger

Winston es un logger diseñado para poder trabajar con multitransportes para nuestra aplicación, utiliza dos conceptos importantes:

- ✓ Transporte: Sistema de almacenamiento de nuestros logs.
- ✓ Nivel : Sistema de prioridad que tiene cada log, para definir si un log tiene autorización para pasar por un transporte.



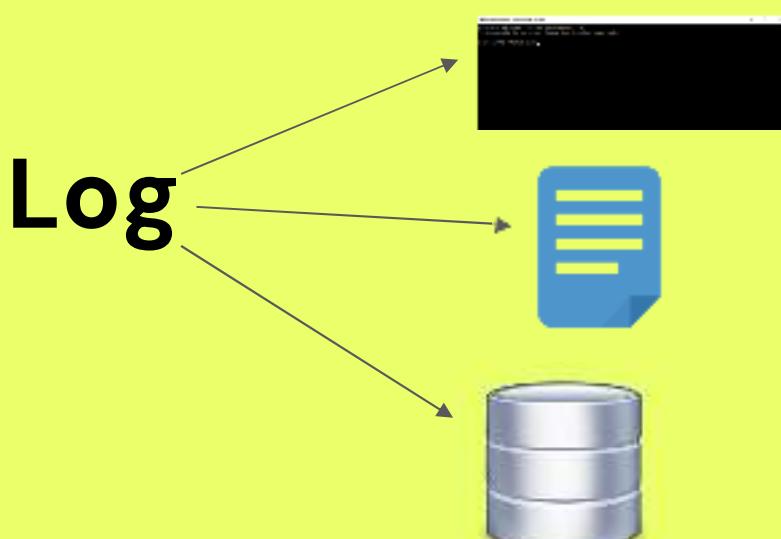
CODERHOUSE

Transportes

Los transportes nativos de Winston permiten que nuestros logs puedan salir de la consola y enviarse por otros medios:

- ✓ Escribirse en un archivo
- ✓ Enviarse a algún servidor externo por http

Con ayuda de su comunidad activa, también pueden enviarse a bases de datos e incluso por mail



Niveles

Establecer un nivel de prioridad para cada mensaje por enviar es crucial.

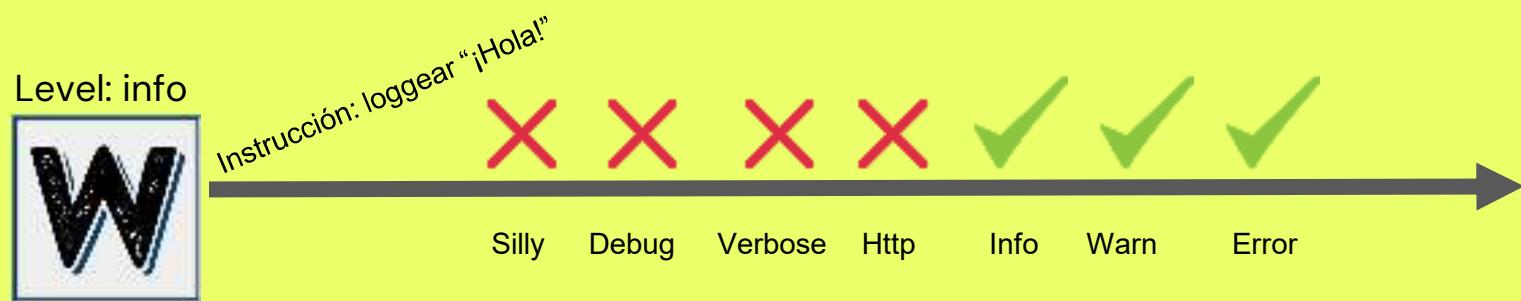
Un nivel al final permitirá hacer saber a nuestro logger “qué es importante mostrar” y “qué podría ignorar en algunos casos”.

A continuación se te presentan los niveles principales utilizados en wintson (Ojo, la prioridad máxima comienza en 0):

```
{  
  error: 0,  
  warn: 1,  
  info: 2,  
  http: 3,  
  verbose: 4,  
  debug: 5,  
  silly: 6  
}
```

¿Cómo funcionan los niveles?

Cuando un logger se configura en un nivel particular, éste podrá loggear **no sólo a dicho nivel**, sino que, además, tendrá la posibilidad de loggear a todos los niveles superiores al configurado en el logger.



Nota: Para lograr loggear en los tres niveles, ocuparemos un logger.info, logger.warn y logger.error respectivamente. (No se muestran los tres en un solo método).

Instalando Winston

Ahora que comprendemos los conceptos principales que necesitamos abordar para trabajar con un logger, es momento de comenzar a trabajar con Winston.

Para ello, utilizaremos el comando:

```
npm install winston
```

Tenemos dos formas de poder trabajar con el logger: o declararlo directamente en nuestro archivo principal (no recomendado), o aislarlo en un archivo externo y configurar un middleware para utilizarlo.

Archivo logger.js

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows the project structure:
 - OPEN EDITORS: logger.js, app.js
 - PRUEBASWINSTON: node_modules, src (expanded), utils (expanded), logger.js (selected), app.js
 - package-lock.json, package.json
- EDITOR:** The file logger.js is open, displaying the following code:

```
logger.js  X  app.js

src> utils > logger.js > addLogger
1 import winston from 'winston';
2
3 const logger = winston.createLogger({
4   /**
5    * A partir de winston.createLogger creamos nuestro logger con los transportes que
6    * necesitamos, en este caso, definimos un transporte de consola para funcionar
7    * solo a partir del nivel http.
8    */
9   transports: [
10     new winston.transports.Console({ level: "http" })
11   ]
12 })
13
14 /**
15  * Ahora, a partir de un middleware, vamos a colocar en el objeto req el logger,
16  * aprovecharemos además para hacer nuestro primer log.
17 */
18
19 export const addLogger = (req,res,next) =>{
20   req.logger = logger;
21   req.logger.http(` ${req.method} en ${req.url} - ${new Date().toLocaleTimeString()}`)
22   next();
23 }
```

Aplicando Middleware

The screenshot shows a code editor interface with the following details:

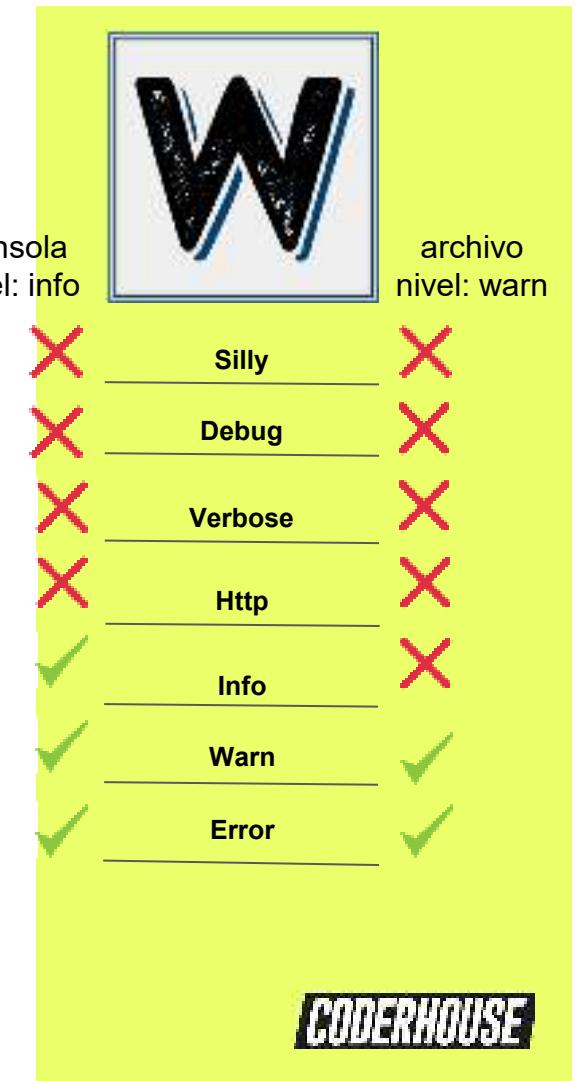
- EXPLORER:** Shows the project structure:
 - OPEN EDITORS: `logger.js`, `app.js`
 - PRUEBASWINSTON: `node_modules`, `src` (expanded), `utils` (expanded), `logger.js` (selected), `app.js`
 - Other files: `package-lock.json`, `package.json`
- Code Editor:** `app.js` file content:

```
1 import express from 'express';
2 import { addLogger } from './utils/logger.js';
3
4 const app = express();
5
6 app.use(addLogger());
7
8 app.get('/', (req, res) => {
9   res.send({message:"¡Prueba de logger!"})
10 }
11
12 app.listen(8080, () => console.log("Listening"));
13 }
```
- TERMINAL:** Log output:

```
{"level":"http","message":"GET en / - function toLocaleTimeString() { [native code] }"}
[nodemon] restarting due to changes...
[nodemon] starting `node src/app.js`
Listening
{"level":"http","message":"GET en / - 23:30:26"}
```

Agregamos un transporte adicional

Parte de la potencia de Winston está en colocar múltiples transportes para trabajar en conjunto. Así, podemos separar los logs “poco importantes” para mostrarse sólo en consola, pero los logs más importantes guardarse en un archivo para poder revisarse posteriormente.

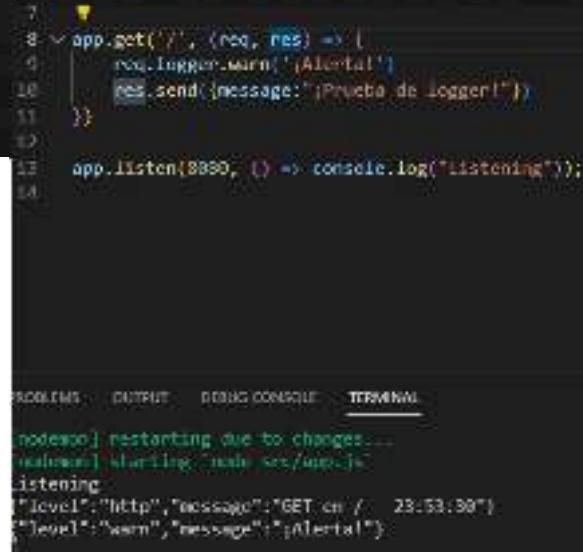


Agregamos el nuevo transporte: File

```
JS logger.js  X  JS app.js
src > utils > JS logger.js > addLogger
1 import winston from 'winston';
2
3 const logger = winston.createLogger({
4   transports: [
5     new winston.transports.Console({ level: "http" }),
6     new winston.transports.File({filename:'./errors.log',level:'warn'})
7   ]
8 })
9
10 /**
11  * Ahora, a partir de un middleware, vamos a colocar en el objeto req el logger,
12  * aprovecharemos además para hacer nuestro primer log.
13 */
14
15 export const addLogger = (req,res,next) =>{
16   req.logger = logger;
17   req.logger.http(` ${req.method} en ${req.url} - ${new Date().toLocaleTimeString()}`)
18   next();
19 }
```

Ahora utilizaremos un logger.warn y veamos el comportamiento

Ejecutamos un "logger.warn" en la petición y vemos cómo se muestra en consola. Esto a causa del primer transporte



```
7 app.get('/', (req, res) => {
8   res.logger.warn('Alerta!');
9   res.send({message:'¡Prueba de logger!'});
10 })
11
12 app.listen(3000, () => console.log("Listening"));
13
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
(node:3001) restarting due to changes...
(node:3001) starting main script/app.js
listening
{"level":"http","message":"GET /  - 23:53:30"}
{"level":"warn","message":"¡Alerta!"}
```

Sin embargo, gracias al segundo transporte, notamos cómo en el archivo también se escribió el log warn. ¡El archivo además ignoró el log de nivel http!



```
errors.log ×
errors.log
1 {"level":"warn","message":"¡Alerta!"}
2
```



Logger multientorno

Duración: 10–15 minutos

CODERHOUSE



ACTIVIDAD EN CLASE

Logger multientorno

Con base en lo aprendido de los loggers

- Configurar el primer logger (devLogger) para que cuente con un transporte Console a nivel Verbose.
- Crear además un logger (prodLogger) para que cuente con un transporte Console a nivel http y un transporte File a nivel warn
- Configurar el middleware que setea el logger en el objeto **req**, para que coloque el devLogger, o el prodLogger según sea el entorno.
- Corroborar los logs en múltiples entornos y analizar el comportamiento.

Agregando nuestra propia configuración

```
logger.js X
src > utils > logger.js > logger > transports > format
1 import winston from 'winston';
2
3 const customLevelsOptions = {
4   levels: [
5     fatal: 0,
6     error: 1,
7     warning: 2,
8     info: 3,
9     debug: 4,
10   ],
11   colors: [
12     fatal: 'red',
13     error: 'orange',
14     warning: 'yellow',
15     info: 'blue',
16     debug: 'white'
17   ]
18 }
```

¿Y si la empresa donde trabajo necesita un nivel de prioridad distinta? Para cada equipo de desarrollo, la importancia de los logs puede variar. Por ejemplo, para un equipo un error puede no ser tan importante, pero un **fatal error** podría ser la perdición.

Entonces, podemos colocar nuestros propios niveles, colores y formatos, según las necesidades del desarrollo que se estén realizando.

Reconfiguremos nuestro logger

Ahora, con los nuevos niveles y colores que deseamos utilizar, podemos colocar la configuración de nuestro logger de la siguiente forma:

```
const logger = winston.createLogger({  
    levels: customLevelOptions.levels, //Aca los niveles se basan en las definiciones  
    transports: [  
        new winston.transports.Console()  
        , {  
            level: "info", //El nivel debe coincidir con nuestro nuevo configuración  
            //El formato solo definirá el que se mostrarán los mensajes de este log.  
            //Algunas de las columnas son más interesantes.  
            format: winston.format.combine()  
            , winston.format.colorize({ values: customLevelOptions.values })  
            , winston.format.simple()  
        }  
    ],  
    exitOnError: false,  
    filename: './errors.log',  
    level: 'warning', //Esto nos muestra 'warning', 'error' y 'error' según lo que escribimos  
    format: winston.format.simple()  
})
```

¡Cuidado al momento de loggear!

Ahora que hemos cambiado los niveles, no podemos utilizar los logger.warn o logger.silly que tenían habitualmente,

Ahora deberemos utilizarlos según lo que definimos, según el ejemplo presentado, nuestros logs serían:

```
logger.fatal()  
logger.error()  
logger.warning()  
logger.info()  
logger.debug()
```

¡Mira cómo queda estilizado!

```
Listening  
info: GET en / - 00:22:41  
warning: warning
```



Break

¡10 minutos y volvemos!

CODERHOUSE

¡Atención!

Recuerda instalar Docker Desktop para la próxima clase.

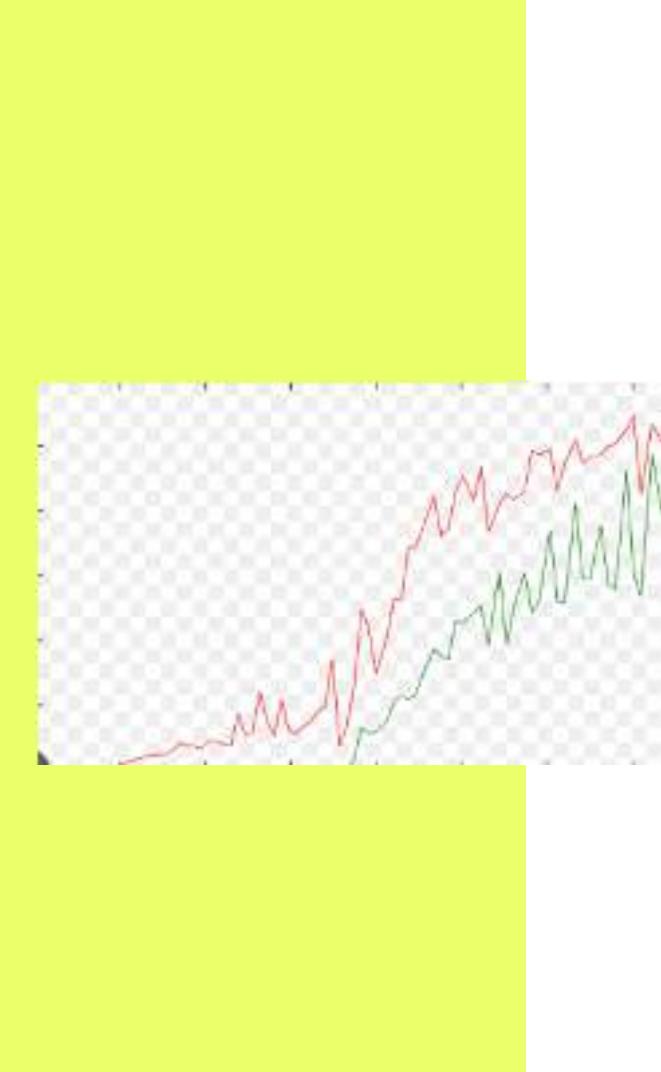


[Página oficial](#)

CODERHOUSE

Testing de performance

CODERHOUSE



¿Mi servidor funcionará en el mundo real?

Nuestro servidor se encuentra perfectamente seguro en nuestro entorno local. Cuando corroboramos si una función realiza lo esperado, hacemos una prueba paso a paso hecha por nosotros.

Sin embargo, cuando mi servidor salga a la luz y comience a ser utilizado por un sinnúmero de internautas, ¿mi servidor podrá soportar mis funciones en un entorno de tráfico elevado?

Simulando carga de peticiones

Es hora de probar nuestro servidor más allá del testing manual que hacemos.

Vamos a utilizar una herramienta que nos permita simular múltiples peticiones a lo largo del tiempo, con el fin de ver qué tal responde mi servidor y qué tan bien tolera funciones complejas con peticiones recurrentes.

Para esto, utilizaremos **Artillery**. A partir de una configuración muy simple, podremos simular **usuarios virtuales**, los cuales emularán las peticiones que necesitamos.

¡Vamos a crear un servidor con algunas operaciones para ver cómo reacciona!

Artillery

CODERHOUSE

Artillery

Es un toolkit de performance que nos permitirá someter a nuestro servidor a pruebas para corroborar la fiabilidad de éste en un entorno real.

Puede configurarse de múltiples maneras para poder simular diferentes entornos, con diferentes peticiones, en diferentes lapsos de tiempo, simulando tráficos reales.

Una vez realizado ésto, nos devolverá un reporte que podemos utilizar para analizar los resultados de dichos escenarios, ayudándonos a corroborar la estabilidad del servidor, o en caso contrario, a tomar decisiones para arreglar los problemas que reporte.



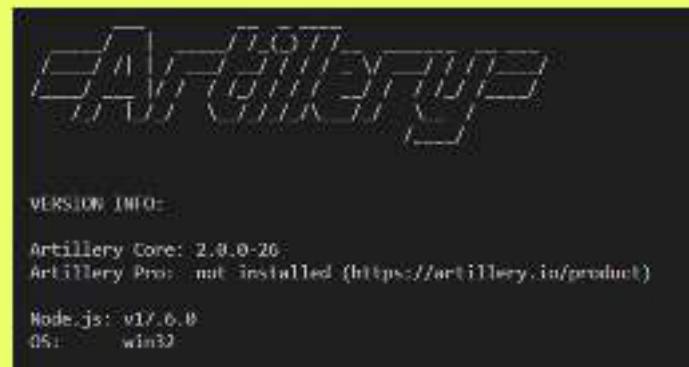
CODERHOUSE

Instalación

Ya que artillery es utilizado para hacer pruebas de manera externa a nuestro servidor, vamos a instalar la dependencia de manera global, utilizando:

```
npm install -g artillery
```

Luego corroboramos que tengamos la librería a partir de **npm artillery -v**. Obtenremos un mensaje similar a éste:



```
Arillery PRO
VERSION INFO:
Artillery Core: 2.9.0-26
Artillery Pro: not installed (https://artillery.io/product)
Node.js: v17.6.0
OS: win32
```

Simulemos algunas operaciones

Para poder ver a Artillery en acción, es necesario tener algunos endpoints que simulen operaciones, para ello, vamos a realizar:

- Una operación sencilla, operación que no debería representar complicaciones ni tiempos complicados de respuesta.
- Una operación compleja, la cual debería demorar una mayor cantidad de tiempo en responder.

```
app.get('/operacionsencilla', (req, res) => {
  let sum = 0;
  for (let i = 0; i < 1000000; i++) {
    sum += i
  }
  res.send({ sum });
}

app.get('/operacioncompleja', (req, res) => {
  let sum = 0;
  for (let i = 0; i < 5e8; i++) {
    sum += i;
  }
  res.send({ sum });
})
```

Ejecutando el comando simple de artillery

Artillery es muy potente, sin embargo también puede evitarnos el realizar operaciones muy complejas, para ello, nos ha dado la posibilidad de utilizar el comando **quick**.
Éste utilizará las siguientes opciones:

- --count: Especifica el número de usuarios virtuales que se crearán para hacer las peticiones
- --num: Especifica el número de peticiones que realizará cada usuario
- -o : Devuelve un formato json con los resultados del test.

```
artillery quick --count 40 --num 50 "http://localhost:8080/operacionseccilla" -o simple.json
```

CODERHOUSE

El test realizado deberá verse así

```
  "aggregates": [],
  "counters": [
    {
      "http_codes_200": 2000,
      "http_responses": 2000,
      "http_requests": 2000,
      "users_failed": 0,
      "users_completed": 48,
      "users_created_by_name": 48,
      "users_created": 48
    },
    {
      "times": [
        {
          "http.request_rate": 77
        }
      ],
      "firstCounterAt": 1668320000000,
      "firstHistogramAt": 1668320000000,
      "lastCounterAt": 1668320000000,
      "lastHistogramAt": 1668320000000,
      "firstMetricAt": 1668320000000,
      "lastMetricAt": 1668320000000,
      "period": 1668320000000,
      "summaries": [
        {
          "http.response_time": {
            "min": 3,
            "max": 246,
            "count": 2000,
            "p50": 144,
            "median": 144,
            "p90": 147,
            "p99": 153,
            "p995": 159.1,
            "p999": 172.5,
            "p9999": 218.6
          }
        },
        {
          "users.session_length": {
            "min": 3001.1,
            "max": 6819.4,
            "count": 48,
            "p50": 6762.6,
            "median": 6762.6,
            "p75": 6762.6,
            "p99": 6838,
            "p999": 6838
          }
        }
      ]
    }
  ]
}
```

Interpretando resultados (1)

```
"counters": {  
    "http.codes.200": 2000,  
    "http.responses": 2000,  
    "http.requests": 2000,  
    "vusers.failed": 0,  
    "vusers.completed": 40,  
    "vusers.created_by_name.0": 40,  
    "vusers.created": 40
```

- ✓ **http.codes.200.** Especifica el número de códigos 200 que se devolvieron, ya que son 40 usuarios, con 50 peticiones cada uno, indica que todas las peticiones se resolvieron sin problema.
- ✓ **vusers.created.** Especifica el número de usuarios de prueba que se generaron, indicando que el test está realizado sobre estos 40 usuarios.

Interpretando resultados (2)

```
"http.response_time": {  
  "min": 3,  
  "max": 246,  
  "count": 2000,  
  "p50": 144,  
  "median": 144,  
  "p75": 147,  
  "p90": 153,  
  "p95": 159.2,  
  "p99": 172.5,  
  "p999": 210.6  
}
```



http.response_time. Indica estándares sobre todas las peticiones realizadas:

- min: la petición más rápida finalizó en 3 milisegundos.
- max: La petición más lenta se resolvió en 246 milisegundos
- median: **NO CONFUNDIR CON PROMEDIO**, indica que la mitad de las peticiones duraron 144 milisegundos o menos.
- **pX**: indica percentiles, medida estadística que señala que un determinado porcentaje demoró **n** tiempo o menos.

Interpretando resultados (3)

```
"vusers.session_length": {  
    "min": 5021.1,  
    "max": 6819.4,  
    "count": 40,  
    "p50": 6702.6,  
    "median": 6702.6,  
    "p75": 6702.6,  
    "p90": 6838,  
    "p95": 6838,  
    "p99": 6838,  
    "p999": 6838  
}
```

✓ **vusers.session_length:** Hace referencia al tiempo que demoró cada usuario virtual a lo largo de toda su trayectoria en el flujo de las peticiones realizadas

- min: El usuario que más rápido finalizó con sus peticiones duró 5 segundos activo
- max: El usuario con más demora en las respuestas de sus peticiones duró casi 7 segundos activo.
- median: **No confundir con promedio.** Indica que la mitad de los usuarios duró 6.7 segundos o menos en la sesión.
- pX: Percentiles, medida estadística.

Ejecutando la operación compleja

Se realizará la misma configuración, esta vez apuntando al endpoint de la operación compleja:

```
artillery quick --count 40 --num 50 "http://localhost:8080/operacioncompleja" -o complejo.json
```

Mismos usuarios (40), con el mismo número de peticiones cada uno (50), debería devolvernos un total de 2000.

Aunque... nuestra operación es bloqueante **y esta vez es un número mucho más elevado**. ¿Será que cambiará mucho el resultado?

El test realizado deberá verse así

```
# ./xperf -f completion.json -o completion
D:\completion.json > D:\completion
1 ~ $ 2 ~ 3 ~ "aggregates": []
4 ~ "counters": [
5 ~ "events-created-by-xperf-0": 48,
6 ~ "events-created": 48,
7 ~ "http.requests": 50,
8 ~ "http.codigo_200": 38,
9 ~ "http.responses": 38,
10 ~ "errors.ETIMEDOUT": 48,
11 ~ "events.failed": 48
12 ~ ]
13 ~ "frames": [
14 ~ "http.request_rate": 2
15 ~ ]
16 ~ "timeCounters": 366327285216,
17 ~ "firstTimestamp": 166327285431,
18 ~ "lastTimestamp": 366327285601,
19 ~ "lastHistogram": 366327285610,
20 ~ "endTimestamp": 366327285116,
21 ~ "startTimestamp": 366327284501,
22 ~ "period": 1000177100000,
23 ~ ]
24 ~ "summaries": [
25 ~ "http.responses_kinet": [
26 ~ "min": 494,
27 ~ "max": 9307,
28 ~ "count": 50,
29 ~ "p50": 2416.0,
30 ~ "median": 2416.0,
31 ~ "p75": 4492.0,
32 ~ "p95": 7557.0,
33 ~ "p99": 7557.0,
34 ~ "p999": 7557.0
35 ~ ]
36 ~ ]
37 ~ "histograms": [
38 ~ "http.responses_kinet": [
39 ~ "min": 494,
40 ~ "max": 9307,
41 ~ "count": 50,
42 ~ "p50": 2416.0,
43 ~ "median": 2416.0,
```

Interpretando resultados (1)

¡Impresionante diferencia! Analicemos lo ocurrido.

```
"vusers.created_by_name.0": 40,  
"vusers.created": 40,  
"http.requests": 50,  
"http.codes.200": 10,  
"http.responses": 10,  
"errors.ETIMEDOUT": 40,  
"vusers.failed": 40
```

- ✓ **vusers.created:** Hasta aquí todo normal, son 40 usuarios.
- ✓ **http.requests:** También aquí, hablamos de 50 requests por usuario
- ✓ **http.codes.200:** ¿Sólo 10 resultados 200? qué ha pasado con las otras 1990 respuestas?
- ✓ **errors.ETIMEDOUT:** Aquí comienza a tener sentido, parece ser que, de los 40 usuarios virtuales, debido a la tardanza de las operaciones, los 40 fueron rechazados por tiempo de espera agotado, es por eso que **vusers.failed** es igual a 40

Interpretando resultados (2)

Veamos entonces a qué se debió este timeout

- ✓ **http.response_time:** Vamos a desglosar los tiempos de respuesta
 - **min:** La petición más rápida demoró casi medio segundo en resolverse.
 - Sin embargo, notamos que la petición más demorada duró **9 segundos para resolverse**, esto ocasionando que todas las peticiones que estuvieran detrás de ésta, fueran rechazadas por esperar demasiado tiempo.
 - **p999:** Hablando de percentiles, vemos cómo las otras peticiones, a pesar de no haber demorado 9 segundos, también respondieron en un tiempo de **7 segundos**. ¡Vaya demora!

```
  "intermediate": [
    {
      "counters": {
        "vusers.created_by_name.0": 40,
        "vusers.created": 40,
        "http.requests": 47,
        "http.codes.200": 7,
        "http.responses": 7
      },
      "rates": {
        "http.request_rate": 10
      },
      "http.request_rate": null,
      "firstCounterAt": 1668327285116,
      "firstHistogramAt": 1668327285631,
      "lastCounterAt": 1668327289735,
      "lastHistogramAt": 1668327289734,
      "firstMetricAt": 1668327285116,
      "lastMetricAt": 1668327289735,
      "period": "1668327280000",
      "summaries": {
        "http.response_time": {
          "min": 494,
          "max": 4466,
          "count": 7,
          "p50": 1939.5,
          "median": 1939.5,
          "p75": 2416.8,
          "p90": 2893.5,
          "p95": 2893.5,
          "p99": 2893.5,
          "p999": 2893.5
        }
      }
    }
  ]
```

¿Obtuvimos un valor “intermediate”?

Los tests pueden demorar una buena cantidad de tiempo, lo cual significa que Artillery entregará reportes “periódicos” durante el tiempo de vida de dicho proceso de testing

Este reporte periódico se realiza cada 10 segundos, lo cual significa que, en este ejemplo particular, de las 10 respuestas 200 que conseguimos recibir, 7 fueron resueltas en los primeros 10 segundos, 3 después de los 10 segundos, y el resto fue rechazado por timeout.

CODERHOUSE

Analizando resultados

Notamos que nos enfrentamos a una cruda realidad: Que un proceso siempre funcione cuando lo ejecutamos una vez, no garantiza su correcto funcionamiento en tráfico elevado.

Debemos realizar una configuración extra para que nuestro servidor pueda soportar muchas más peticiones recurrentes y así poder realizar tareas “multitask”

Más adelante veremos cómo realizar este “potenciamiento” con el fin de que nuestro servidor se mantenga más sólido y procese mejor las peticiones, a este proceso lo entenderemos como “escalabilidad”.

¡Vamos a ver qué más podemos realizar con artillery!

Testing avanzado de performance

CODERHOUSE



Un comando no siempre es suficiente.

Imagina que tienes que testear un flujo de registro con login. Al revisar nuestro comando **artillery quick**, notamos una limitante: **sólo podemos testear un endpoint**. Además de sólo servir para datos que no necesiten consistencia.

Para ello, Artillery nos permite realizar un flujo de testing mucho más avanzado, donde podemos simular **escenarios**, es decir, contextos completos en los cuales podemos probar si el flujo fue aceptado en su totalidad.

CODERHOUSE

Artillery al siguiente nivel

Para poder realizar pruebas más complejas, necesitaremos un archivo de configuración de la prueba.

Con un archivo de configuración podemos tener un mejor control sobre la prueba que queremos realizar

Algunas de las cosas que podemos realizar con el flujo son:

- ✓ Pruebas de peticiones a múltiples endpoints.
- ✓ Simulación de flujos de peticiones.
- ✓ Tiempos de espera
- ✓ Envío de parámetros
- ✓ Guardado de parámetros de peticiones previas para utilizar en otros flujos.
- ✓ Tomar funciones de contextos de funciones generadoras.
- ✓ ¡Y mucho más!



Ejemplo en vivo

El profesor actualmente cuenta con un proyecto de autenticación (registro y login) simple con mongodb. Se realizará un archivo de configuración de artillery para simular un flujo en el que el usuario se registra y luego se loguea.

Duración: **15 minutos**

CODERHOUSE

Ejemplo: Entendiendo el código que ya tenemos (register y login)

Se debe tener un sistema de registro y login muy básico con funciones que permitan crear y leer usuarios desde una base de datos de MongoDB. Se muestra el ejemplo de un controlador de sesiones que cuenta con estos dos métodos:

```
 0 480 16 0 sessionController.js
 1 // controllers/sessionController.js
 2
 3 import UserModel from '../models/User.js';
 4
 5 const express = require('express');
 6 const router = express.Router();
 7
 8
 9 // Register
10 router.post('/register', async (req, res) => {
11   const { firstName, lastName, email, password } = req.body;
12   console.log(`Registerando ${firstName} ${lastName} - ${email} con pw: ${password}`);
13   try {
14     const user = await UserModel.create({ ...req.body });
15     res.status(201).send({ status: 'ok', user });
16   } catch (error) {
17     res.status(400).send({ status: 'error', error: 'Email taken' });
18   }
19 }
20
21 // Login
22 router.post('/login', async (req, res) => {
23   const { email, password } = req.body;
24   try {
25     const user = await UserModel.findOne({ email });
26     if (!user || !user.authenticate(password)) {
27       return res.status(401).send({ status: 'error', error: 'Email or password incorrect' });
28     }
29     const token = jwt.sign({ user }, process.env.JWT_SECRET);
30     res.cookie('token', token, { maxAge: 60 * 60 * 24 * 30, httpOnly: true });
31     res.status(200).send({ status: 'ok', message: 'Logged in!' });
32   } catch (error) {
33     res.status(500).send({ status: 'error', error: 'Internal server error' });
34   }
35 }
```

Ejemplo: Entendiendo el código que ya tenemos (faker y test user)

Además en nuestro archivo principal app, deberemos contar con un endpoint que nos permita generar un usuario de prueba con los datos:

- first_name
- last_name
- email
- password.

Estos cuatro valores se envían en la respuesta

```
//Este endpoint sirve para poder crear el usuario virtual con variables para utilizar en el resto de endpoints
app.get('/api/test/user',(req,res)=>{
  let first_name = faker.name.firstName();
  let last_name = faker.name.lastName();
  let email = faker.internet.email();
  let password = faker.internet.password();
  res.send({first_name,last_name,email,password})
```

Ejemplo: Entendiendo el código que ya tenemos (package.json)

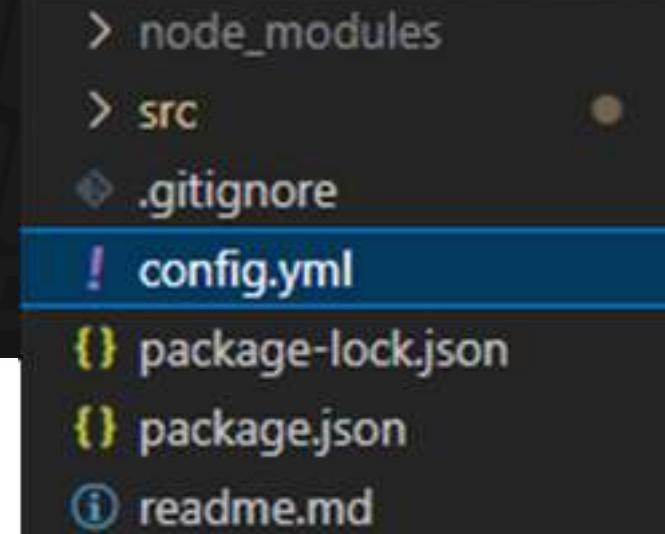
Nuestro package.json contará con las siguientes dependencias:

- Faker para generar el usuario test
- bcrypt para hashear las contraseñas y compararlas.
- mongoose para la conexión y los modelos de la base de datos
- express como servidor principal.
- **artillery** (instalado en -D) para leer el test.
- **artillery-plugin-metrics-by-endpoint**, para poder separar los tiempos y métricas por cada endpoint visitado en el flujo de testing.

```
13 "dependencies": {  
14   "@faker-js/faker": "^7.6.0",  
15   "artillery-plugin-metrics-by-endpoint": "^1.0.2",  
16   "bcrypt": "^5.1.0",  
17   "express": "^4.18.2",  
18   "mongoose": "^6.7.2"  
19 },  
20 "devDependencies": {  
21   "artillery": "^2.0.0-26"  
22 }
```

Ejemplo: creando un archivo config.yml

A la altura de nuestro proyecto, creamos un archivo config.yml, donde escribiremos todas las instrucciones de testing de nuestro respectivo flujo de performance con artillery.



Ejemplo: primera parte del archivo.

La primera parte del archivo consiste en la configuración, donde seteamos los plugins (como las métricas que instalamos), el target (servidor a hacer las peticiones). y las phases.

Las fases funcionan de la siguiente forma:

Se generarán **10 usuarios virtuales** (arrivalRate), cada segundo, por **un lapso de 20 segundos** (duration)

Esto significa que se generarán 200 usuarios en total para probar nuestro flujo.

```
JS app.js M ! config.yml X
! config.yml > [ ]scenarios > {} 0 > [ ]flow > {} 1
1   config:
2     plugins:
3       metrics-by-endpoint: {}
4     target: 'http://localhost:8080'
5     phases:
6       - duration: 20
7         arrivalRate: 10
```

Ejemplo: Segunda parte del archivo

La segunda parte es el escenario, éste define el flujo que va a tener cada petición, nota cómo definimos los gets y posts correspondientes a las peticiones /test/user (para generar el usuario de prueba), register y login

```
 5 app.js   f config.yml x
 6 config.yml > []username > () > []flow > () > () get > []register > () > []
 7   - duration: 20
 8   - invocations: 10
 9   scenarios:
10     - name: "Sessions flow (register + login)"
11       flow:
12         - log: "Creando las variables para el usuario de prueba"
13         - get:
14           url: '/api/test/user'
15           description: 'Este paso significa que queremos enviar los campos de la respuesta y guardarlos para futuras peticiones'
16           json:
17             - first_name: "$.first_name"
18               use: "First name"
19             - last_name: "$.last_name"
20               use: "Last name"
21             - email: "$.email"
22               use: "Email"
23             - password: "$.password"
24               use: "Password"
25         - post:
26           url: "/api/sessions/register"
27           json:
28             description: 'Este paso significa que queremos enviar los datos en el body de la petición'
29             first_name: "{{.first_name}}"
30             last_name: "{{.last_name}}"
31             email: "{{.email}}"
32             password: "{{.password}}"
33         - think:
34           description: 'Este paso significa que el usuario espera dos segundos entre la register y el login.'
35           log: "Login user"
36         - post:
37           url: "/api/sessions/login"
38           json:
39             email: "{{.email}}"
40             password: "{{.password}}"
```

Ejemplo: Segunda parte del archivo

La segunda parte es el escenario, éste define el flujo que va a tener cada petición, nota cómo definimos los gets y posts correspondientes a las peticiones /test/user (para generar el usuario de prueba), register y login

```
 5 app.js   f config.yml x
 6 config.yml > []username > () > []flow > () > () get > []register > () > []
 7   - duration: 20
 8   - invocations: 10
 9   scenarios:
10     - name: "Sessions flow (register + login)"
11       flow:
12         - log: "Creando las variables para el usuario de prueba"
13         - get:
14           url: '/api/test/user'
15           description: 'Este paso significa que queremos enviar los campos de la respuesta y guardarlos para futuras peticiones'
16           json:
17             - first_name: "$.first_name"
18               use: "First name"
19             - last_name: "$.last_name"
20               use: "Last name"
21             - email: "$.email"
22               use: "Email"
23             - password: "$.password"
24               use: "Password"
25         - post:
26           url: "/api/sessions/register"
27           json:
28             description: 'Este paso significa que queremos enviar los datos en el body de la petición'
29             first_name: "{{.first_name}}"
30             last_name: "{{.last_name}}"
31             email: "{{.email}}"
32             password: "{{.password}}"
33         - think:
34           description: 'Este paso significa que el usuario espera dos segundos entre la register y el login.'
35           log: "Login user"
36         - post:
37           url: "/api/sessions/login"
38           json:
39             email: "{{.email}}"
40             password: "{{.password}}"
```

Ejemplo: Ejecutando el archivo

```
artillery run config.yml --output testPerformance.json
```

Logs del archivo yml

```
Creamos las variables para el usuario de prueba
Registrando al usuario
\ Login user
Login user
| Login user
Login user
- Creamos las variables para el usuario de prueba
Registrando al usuario
Creamos las variables para el usuario de prueba
Registrando al usuario
\ Creamos las variables para el usuario de prueba
Registrando al usuario
Creamos las variables para el usuario de prueba
Registrando al usuario
Login user
```

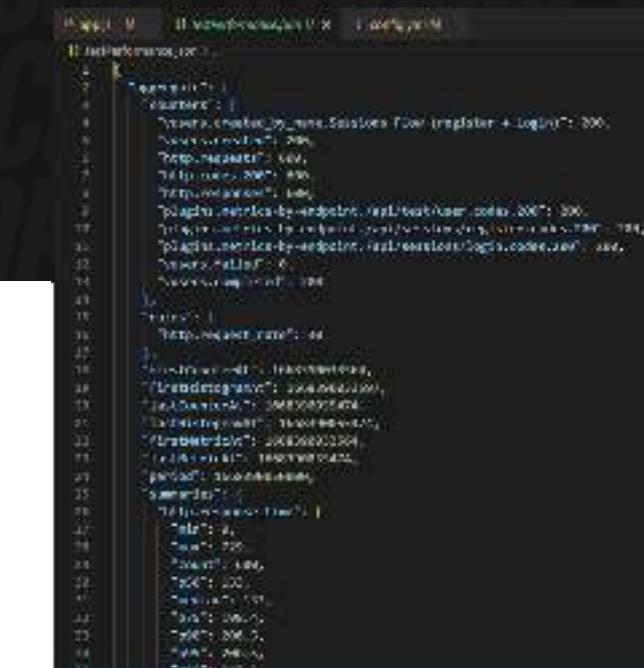
Logs de la terminal del servidor

```
El ingreso de Fritz.Stokes@gmail.com fue satisfactorio
Registering Otto Schumm email: Owen.Fritsch31@hotmail.com and pwd: YSTVCu7eYDZFuiE
El ingreso de Florence_Rippin@yahoo.com fue satisfactorio
Registering Nikita Bogan email: Autumn38@gmail.com and pwd: wCCcIVQUfAjdhY
El ingreso de Giles.Skiles11@hotmail.com fue satisfactorio
Registering Sidney Jones email: Rosario_OHara@gmail.com and pwd: jTdQH01IgiEwMng
El ingreso de Jodie5@gmail.com fue satisfactorio
El ingreso de Bailee76@gmail.com fue satisfactorio
El ingreso de German.Halvorson3@gmail.com fue satisfactorio
El ingreso de Ethyl.Davis@hotmail.com fue satisfactorio
El ingreso de Erick59@hotmail.com fue satisfactorio
El ingreso de Everardo70@gmail.com fue satisfactorio
Registering Deondre Graham email: Rosalinda0@gmail.com and pwd: dTnYAHWjWkbqb
Registering Craig Abshire email: Gustave_Walker76@hotmail.com and pwd: xvChXISuK_zeHfE
Registering Maeve Swaniawski email: Pansy.OConnell@yahoo.com and pwd: Jx6BqDsxCpIBYQ
```

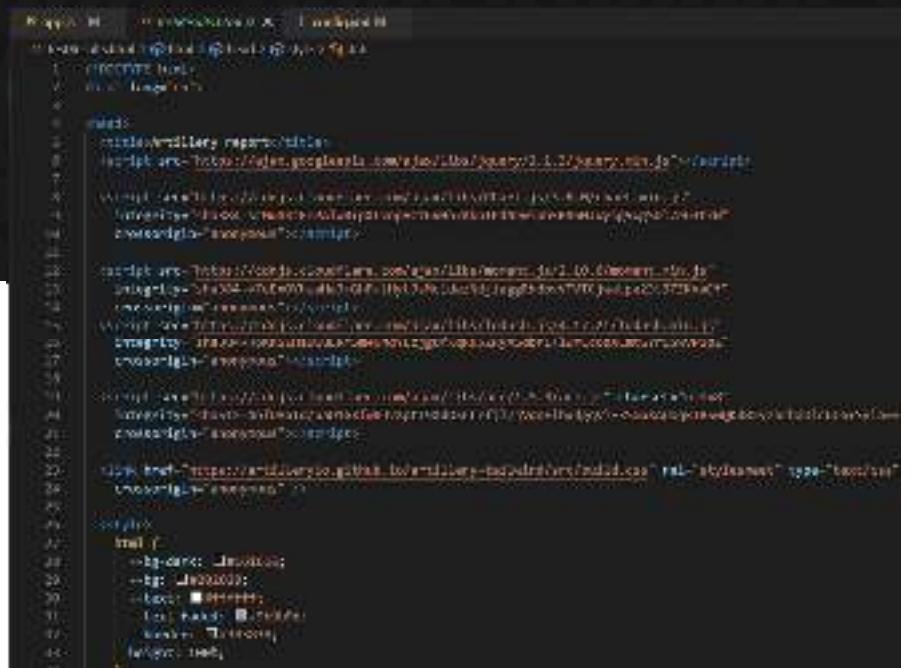
CODERHOUSE

Ejemplo: Generando un reporte

```
artillery report testPerformance.json -o testResults.html
```

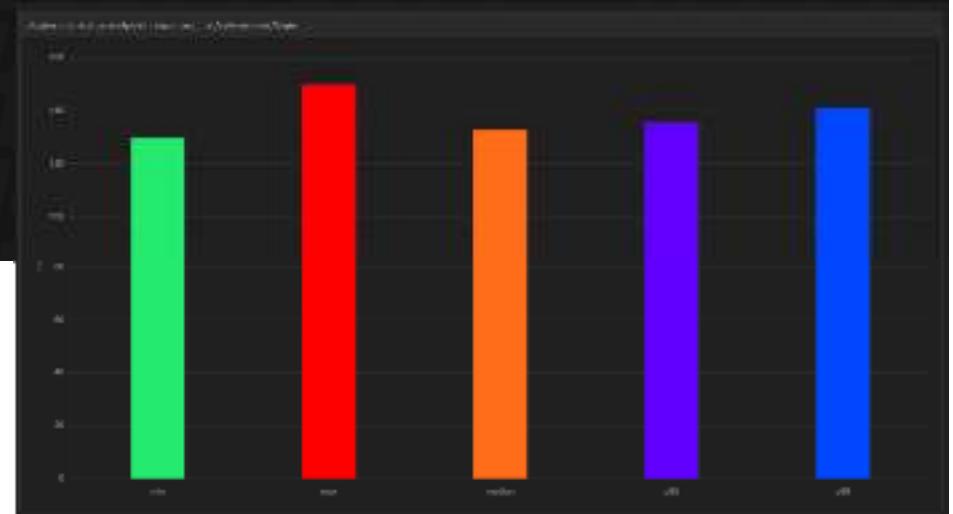


```
1. [root@ip-172-31-10-16 ~]# artillery report testPerformance.json -o testResults.html
2. artillery report...[...]
```



```
1. [root@ip-172-31-10-16 ~]# curl -s http://172.31.10.16/testResults.html | less
2. artillery report...[...]
```

Ejemplo: Visualizando reporte



¡Atención!

Recuerda instalar Docker para la próxima clase.
[¿Qué estoy por instalar?](#)



[Ver tutorial](#)

CODERHOUSE



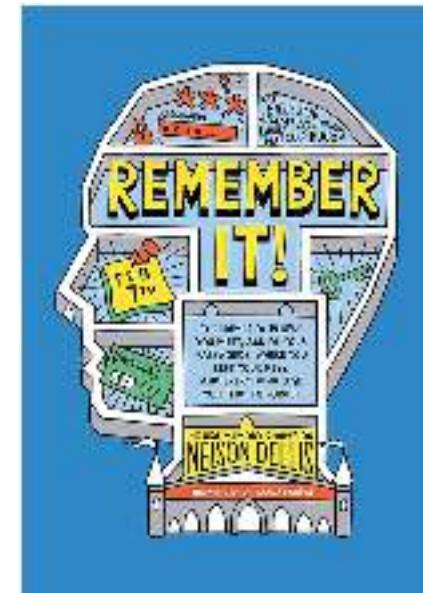
Implementación de logger

Recordemos...

**Basado en el entregable
de la clase 2**

Revisamos y analizamos el
proyecto Adoptme.

Realizamos un servicio de mocking
en dicho proyecto.



Ahora... Agregaremos un logger para tener mejores registros



ACTIVIDAD PRÁCTICA

Implementación de logger

Consigna

- ✓ Basado en nuestro [proyecto principal](#), implementar un logger.

Aspectos a incluir

- ✓ Primero, definir un sistema de niveles que tenga la siguiente prioridad (de menor a mayor):
`debug, http, info, warning, error, fatal`
- ✓ Después implementar un logger para desarrollo y un logger para producción, el logger de desarrollo deberá loggear a partir del nivel debug, sólo en consola

- ✓ Sin embargo, el logger del entorno productivo debería loggear sólo a partir de nivel info.
- ✓ Además, el logger deberá enviar en un transporte de archivos a partir del nivel de error en un nombre “errors.log”
- ✓ Agregar logs de valor alto en los puntos importantes de tu servidor (errores, advertencias, etc) y modificar los `console.log()` habituales que tenemos para que muestren todo a partir de winston.
- ✓ Crear un endpoint “/loggerTest” que permita probar todos los logs

CODERHOUSE



ACTIVIDAD PRÁCTICA

Implementación de logger

Formato

- ✓ link al repositorio de Github con el proyecto sin node_modules

Sugerencias

- ✓ Puedes revisar el testing del entregable [Aquí](#)
- ✓ La ruta “/loggerTest” es muy importante para que tu entrega pueda ser calificada de manera rápida y eficiente. ¡No olvides colocarla!

¿Preguntas?

CODERHOUSE

Resumen de la clase hoy

- ✓ NVM
- ✓ Yarn
- ✓ Conceptos adicionales de npm
- ✓ Creación de dependencia con npm

**Opina y valora
esta clase**

CODERHOUSE

Muchas gracias.

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 5. Testing y Escalabilidad Backend

Clusters & Escalabilidad

CODERHOUSE

Temario

4

Logging y Testing de performance

- ✓ Loggers
- ✓ Testing de performance
- ✓ Testing avanzado de performance

5

Clusters & Escalabilidad

- ✓ Módulo Cluster
- ✓ Docker
- ✓ Docker como PM

6

Orquestación de contenedores

- ✓ DockerHub
- ✓ Orquestación de contenedores
- ✓ Orquestación con Kubernetes

Objetivos de la clase

- **Entender y aplicar** el módulo de Cluster de Nodejs
- **Conocer** Docker
- **Implementar** Docker como un Process Manager

CLASE N°4

Glosario

Artillery: Es un toolkit de performance que prueba nuestro servidor y corrobora su fiabilidad en un entorno real.

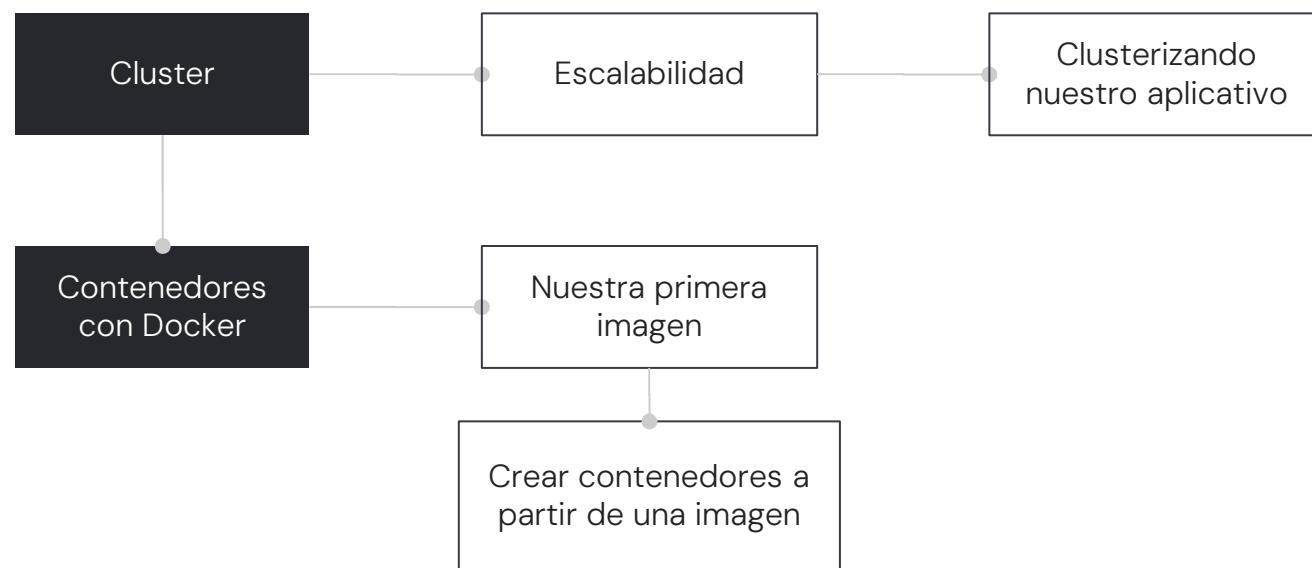
Niveles: permitirá priorizar “qué es importante mostrar” y “qué podría ignorar en algunos casos

Transportes: Los transportes nativos de Winston permiten que nuestros logs puedan salir de la consola y enviarse por otros medios.

Winston Logger: Winston es un logger diseñado para poder trabajar con multitransportes.



MAPA DE CONCEPTOS



Cluster

CODERHOUSE

Recordemos...

¿Recuerdas cómo intentamos hacer un test de performance con Artillery, aplicando “una operación sencilla” y “una operación compleja”?

Al hacer las operaciones complejas, se rechazaron muchas peticiones debido a que el servidor se saturaba y demoraba mucho en responder.

Seguramente en ese momento te enfrentaste a un reto de la realidad:
El servidor no puede atender todas las peticiones recurrentes, mucho menos estando con las manos tan ocupadas con operaciones bloqueantes.

¡Pero las cosas no se pueden quedar así! ¡Deberíamos poder hacer algo! 🚀

CODERHOUSE

Estrategia: Escalabilidad

Cuando hablamos de “escalar” un servidor, lo hacemos a partir de dos conceptos:

- ✓ **Escalamiento vertical:** Mi servidor necesita ser más potente y necesito mejorar el hardware para tener un servidor más potente.
- ✓ **Escalamiento horizontal:** Dividamos las tareas en multi-instancias de servidores que alojen el aplicativo y se apoyen en las tareas complejas.

Escalabilidad vertical

Básicamente, significa **mejorar el hardware** del servidor, para que sea más potente, mucho más rápido y pueda atender una mayor cantidad de peticiones y, por lo tanto, mejorar el performance de los aplicativos.

👉 El escalamiento vertical requiere de grandes inversiones de recursos por parte de las empresas para poder contar con los equipos más actualizados posibles en el mundo de la tecnología.

Además, llegará un punto en el que alcanzaremos un **tope tecnológico**, y tendremos que esperar a que se desarrollen mejores soluciones de hardware para poder comprarlas (un tiempo de espera que una empresa difícilmente puede contener).

Escalabilidad horizontal

Este modelo es más complejo, pero mucho más interesante y eficiente.

La escalabilidad horizontal significa utilizar múltiples servidores, conocidos como **nodos**, los cuales trabajarán en equipo para resolver un problema en particular.

A esta red de **nodos** trabajando juntos, se le conoce como **cluster**, haciendo referencia a que estos múltiples servidores se encuentran en un contexto general donde todos conocen cómo ayudarse a las tareas más complejas.

Así, la diferencia radica en que, cuando necesitamos más recursos, **no hace falta tirar el servidor que ya tenemos a la basura para comprar uno mejor**, sino que podemos conectar otra instancia de otro servidor para que se una a la red de **nodos** y forme parte del **cluster**.

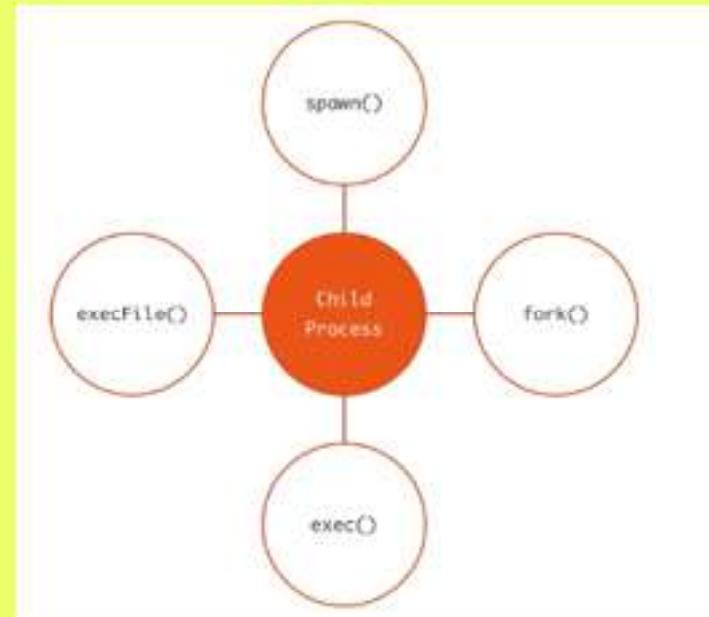


CODERHOUSE

¿Cómo clusterizar nuestro aplicativo?

Para poder configurar satisfactoriamente nuestro servidor a partir de un modelo horizontal, tenemos que recordar cómo funcionaba la gestión de los child process vistos en la clase 0.

¿Recuerdas cómo funcionaba?



Sobre el process id

Cuando un proceso se ejecuta, este tiene dentro de sus características principales una propiedad conocida como **pid**.

Este processId es muy importante para poder trabajar con otros procesos.

Cuando un proceso padre instanciaba un proceso hijo, este mantiene una referencia a partir del **pid**, haciéndole saber que ese proceso es parte de él.

`process.pid`

Sobre el forkeo

Ahora, el proceso global podía generar el nuevo proceso a partir de 4 métodos principales, donde nosotros tuvimos la posibilidad de hablar sobre el forkeo.

La palabra **fork** será clave para hacer referencia a que un proceso nuevo surgirá, pero se mantendrá ligado al proceso que lo generó.

Anteriormente, llamábamos global process al proceso padre que **forkeaba** al proceso hijo.

Sin embargo, esta vez conoceremos al proceso principal como **Primary process** (anteriormente llamado **Master**), mientras que a las múltiples instancias que se generen se llamarán **workers**.

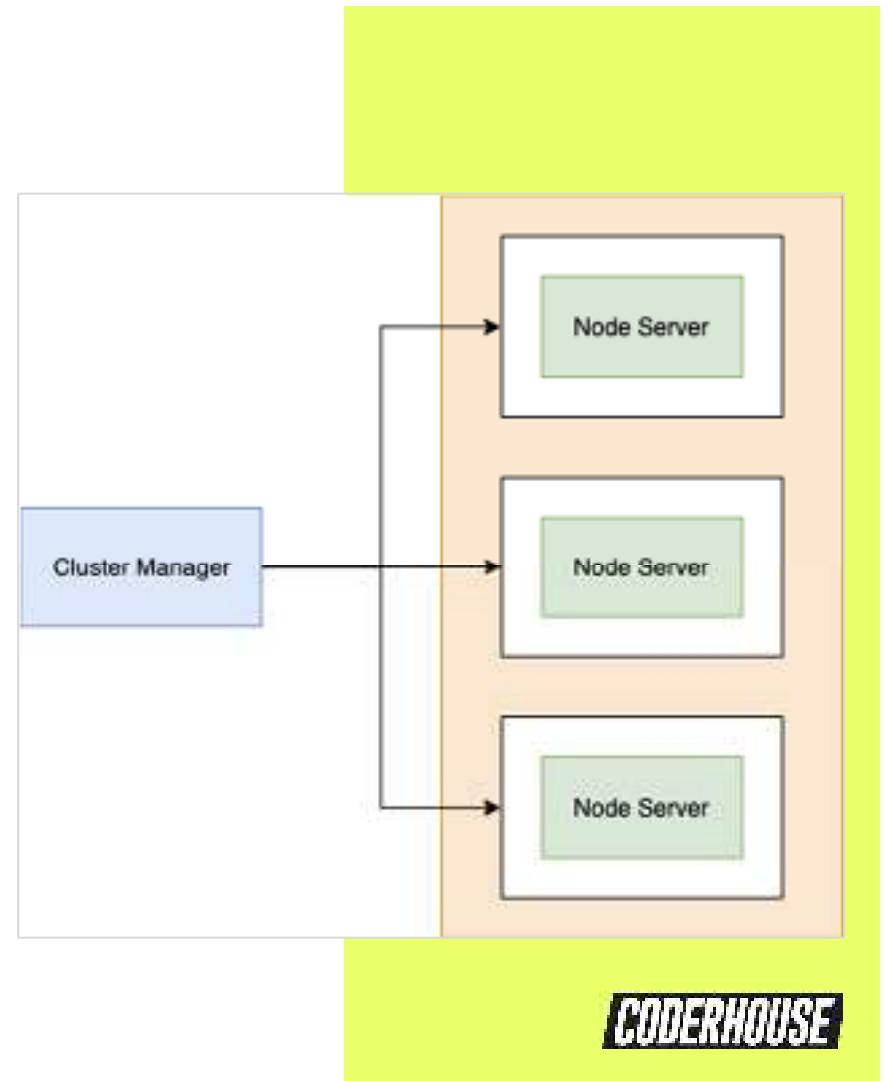
Clusterizando nuestro aplicativo

CODERHOUSE

Módulo nativo cluster

Cluster es un módulo nativo de node.js que nos permitirá ejecutar este concepto de clusterización que recién comentamos, donde podremos tener a un proceso principal contando con un grupo de procesos trabajadores.

Estos trabajadores van a trabajar en conjunto para resolver el problema de las situaciones de las peticiones.



isPrimary?

Vamos a importar el módulo nativo cluster en nuestro archivo app.js y analizaremos primero nuestra primera propiedad principal: **isPrimary**

Esta propiedad nos ayuda a corroborar si el proceso es el principal, o viene forkeado de algún proceso superior.

Cuidado: si tu versión de node es 16+, podrás utilizar isPrimary sin problema, si es anterior deberás utilizar **isMaster**

```
JS app.js    ×  () package.json
src > JS app.js
1   import cluster from 'cluster';
2
3
4   console.log(cluster.isPrimary);
```

true

Realizamos un forkeo desde cluster

Para poder generar nuestro primer proceso trabajador, vamos a hacerlo solo desde el proceso principal, entonces, la lógica es:

- ✓ Si eres un proceso primario, entonces indica que eres el principal y forkea a un trabajador.
- ✓ Si eres un proceso trabajador, entonces indica que eres trabajador y procede a realizar las tareas que corresponden.

Diferenciando primary de workers

```
JS app.js    X  () package.json
src > JS app.js
1 import cluster from 'cluster';
2
3 if(cluster.isPrimary){
4     console.log("Proceso primario, generando proceso trabajador");
5     cluster.fork();
6 }
7 else{
8     console.log("Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!")
9 }
10 }
```

Notamos cómo primero se comunica el proceso primario, pero al realizar el forkeo, se comunica el proceso secundario.

```
Proceso primario, generando proceso trabajador
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
```

Aprovechando la capacidad de un computador

Sabemos que el procesamiento de nuestro servidor será siempre single-threaded. Al realizar nuestros primeros forkeos, estamos comenzando a romper el paradigma que implica.

¿Cómo levantar múltiples instancias, sin que afecte demasiado en tamaño?

Para ello, lo primero debería ser determinar el número de hilos que podrán procesar estos multiprocesamientos, esto lo conseguiremos con las siguientes líneas

```
import { cpus } from 'os';

const numeroDeProcesadores = cpus().length;
console.log(numeroDeProcesadores);
```

En el caso de este ejemplo, trabajaremos con



Generando múltiples trabajadores

```
8 app.js  x  0 packages
src > 8 app.js > ...
1 import cluster from 'cluster';
2 import { cpus } from 'os';
3
4 const numeroDeProcesadores = cpus().length;
5
6 if(cluster.isPrimary){
7   console.log("Proceso primario, generando numeros Trabajadores");
8   for( let i = 0; i<numeroDeProcesadores;i++){
9     cluster.fork();
10   }
11 }
12 else{
13   console.log("Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!")
14   console.log(`Me presento, soy un proceso worker con el id : ${process.pid}`);
15 }
16 }
```

```
!proceso primario, generando proceso trabajador
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 18344
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 14824
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17422
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 14836
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 18272
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17882
Me presento, soy un proceso worker con el id : 18337
Al ser un proceso forkado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
```

¿Por qué tener múltiples procesos?

```
const maximoProcesos = process.argv.length - 1;

if(clusters > 1) {
    console.log(`Proceso ${process.pid}, generando proceso hijo ${clusters}`);
    for(let i = 0; i < maximoProcesos; i++) {
        cluster.fork();
    }
}

else{
    console.log(`Al ser un proceso hijo, no cuenta como primario, por lo tanto la info`);
    console.log(`No es necesario que un proceso mire con su pid: ${process.pid}`);
    const app = express();
}

app.get('/',(req,res) =>
    res.send({status:"success", message:"Petición creada por un proceso worker"})
);

app.listen(8000, ()=>console.log(`Listening on 8000`));

```

```
Mi proceso, soy un proceso worker con el id : 10012
Listening on 8000
Al ser un proceso hijo, no cuenta como primario, por lo tanto listening=false, (entonces soy un worker)
Mi proceso, soy un proceso worker con el id : 10012
Listening on 8000
Listening on 8000
Al ser un proceso hijo, no cuenta como primario, por lo tanto listening=false, (entonces soy un worker)
Mi proceso, soy un proceso worker con el id : 10012
Listening on 8000
Listening on 8000
Al ser un proceso hijo, no cuenta como primario, por lo tanto listening=false, (entonces soy un worker)
Mi proceso, soy un proceso worker con el id : 10012
Listening on 8000
Al ser un proceso hijo, no cuenta como primario, por lo tanto listening=false, (entonces soy un worker)
Mi proceso, soy un proceso worker con el id : 10012
Listening on 8000
Listening on 8000
```

¡Entonces viene lo maravilloso! ¿Recuerdas que comentamos la importancia de que todos estuvieran conectados en un contexto general? ¡Nota cómo lo utilizamos para que cada quien cuente con una instancia del servidor!

CODERHOUSE

¿Qué ocurrió?

En este momento hay múltiples procesos escuchando y trabajando sobre el servidor, en realidad, cada uno es una instancia de servidor.

¿Cómo reconocemos que esto sea cierto?

Vamos a utilizar un comando muy útil en windows conocido como

tasklist /fi "ImagenName eq node.exe"

El comando en negritas significa que encuentre una imagen (instancia de proceso) del proceso **node.exe**

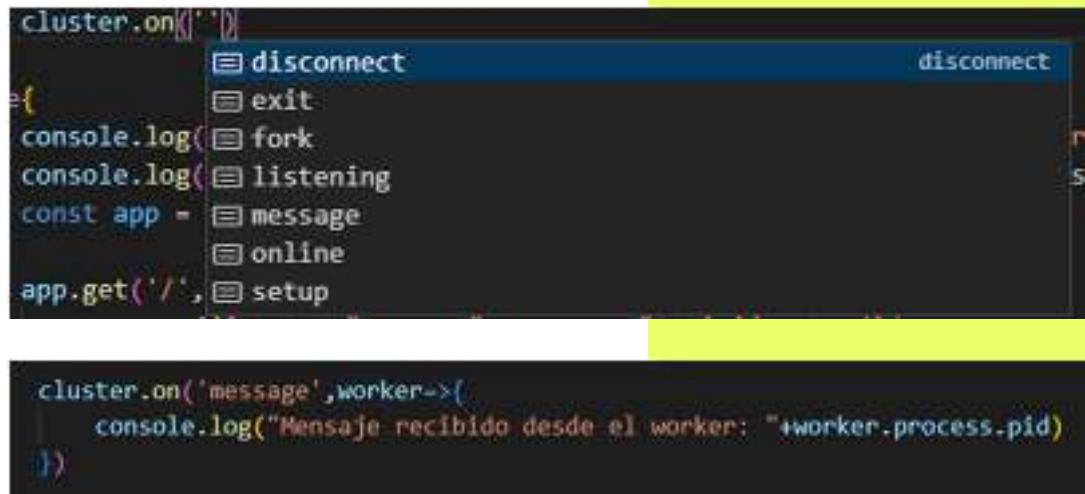
Al ejecutarlo, podemos ver los procesos que corresponden a nuestros respectivos contextos.

Nombre de imagen	PID	Nombre de sección	Nºm. de ses.	Uso de memoria
node.exe	5344	Console	1	31,692 KB
node.exe	16724	Console	1	31,596 KB
node.exe	9412	Console	1	31,532 KB
node.exe	3748	Console	1	31,388 KB
node.exe	16488	Console	1	31,388 KB
node.exe	4916	Console	1	31,408 KB
node.exe	18788	Console	1	31,768 KB
node.exe	15798	Console	1	31,528 KB
node.exe	18412	Console	1	31,596 KB
node.exe	18112	Console	1	31,598 KB
node.exe	12718	Console	1	31,488 KB
node.exe	16712	Console	1	31,788 KB
node.exe	312	Console	1	31,376 KB
node.exe	12278	Console	1	31,852 KB
node.exe	1478	Console	1	31,816 KB
node.exe	17248	Console	1	31,528 KB
node.exe	17488	Console	1	31,688 KB

¡Hay más!

No solo generamos múltiples instancias del servidor, sino que podemos escuchar a eventos desde el proceso primario.

Entonces, seguro te recuerda al proceso de comunicación de socket.io, ya que en el proceso padre podremos colocar **.on** para saber qué pasa con alguno de sus trabajadores.



```
cluster.on('message', worker=>{
  console.log("Mensaje recibido desde el worker: " + worker.process.pid)
})
```

The screenshot shows a code editor with two snippets of Node.js code. The top snippet is part of a file named 'index.js' and contains the following code:

```
cluster.on('message', worker=>{
  console.log("Mensaje recibido desde el worker: " + worker.process.pid)
})
```

The bottom snippet is part of a file named 'worker.js' and contains the following code:

```
cluster.on('message', worker=>{
  console.log("Mensaje recibido desde el worker: " + worker.process.pid)
})
```

¡A esto nos referíamos!

Nota que el proceso primario no solo se está encargando de levantar múltiples procesos, sino que también nos permitirá tener un control sobre estos. Ahora vamos a ponerlo a prueba con las operaciones que ya conocemos.

```
else{
    console.log("Al ser un proceso forjado, no cuento como primario, por lo tanto isPrimary=false. (Entonces soy un worker!)");
    console.log(`Me presento, soy un proceso worker con el id: ${process.pid}`);
    const app = express();

    app.get('/operacionSimple',(req,res)=>{
        let sum = 0;
        for(let i = 0; i<10000;i++){
            sum+=i;
        }
        res.send({status:'success', message:`El worker ${process.pid} ha atendido esta petición, el resultado es ${sum}`})
    });

    app.get('/operacionCompleja',(req,res)=>{
        let sum = 0;
        for(let i = 0; i<5000;i++){
            sum+=i;
        }
        res.send({status:'success', message:`El worker ${process.pid} ha atendido esta petición, el resultado es ${sum}`})
    });
}
```

Ejecutando el proceso de Artillery para las operaciones sencillas

```
artillery quick --count 40 --num 50 "http://localhost:8080/operacionSencilla" -o resultadosSencillos.json
```

```
,  
  "histograms": {  
    "http.response_time": {  
      "min": 0,  
      "max": 13,  
      "count": 2000,  
      "p50": 7,  
      "median": 7,  
      "p75": 7.9,  
      "p90": 8.9,  
      "p95": 10.9,  
      "p99": 12.1,  
      "p999": 13.1  
    },
```

¡Impresionantes resultados! Nota que ahora el tiempo mínimo de resolución de la operación sencilla es tan bajo que Artillery lo ha dejado en 0. y el tiempo máximo es de 13 milisegundos (casi nada)

La operación sencilla era fácil de afrontar, pero ahora falta el verdadero reto: La operación compleja sólo nos permitió finalizar 10 de la meta de 2000 peticiones, veamos cómo se comporta en esta ocasión

Ejecutando el proceso de Artillery para las operaciones complejas

```
artillery quick --count 40 --num 50 "http://localhost:3000/operacionCompleja" -o resultadosComplejos.json
```

The terminal window shows the command to run the test and the resulting JSON file. The JSON file contains the following data:

```
JS app.js          () resultadosComplejos.json ×  () package.json
() resultadosComplejos.json > () aggregate > () counters:
1  {
2    "aggregate": {
3      "counters": [
4        "vusers.created_by_name.0": 40,
5        "vusers.created": 40,
6        "http.requests": 2000,
7        "http.codes.200": 2000,
8        "http.responses": 2000,
9        "vusers.failed": 0,
10       "vusers.completed": 40
11     ],
12     "rates": [
13       "http.request_rate": 5
14     ]
15   }
16 }
```

Voilà! Muy seguramente tu computadora ha tenido que enfrentar una dura y larga batalla para resolver las operaciones complejas, sin embargo, notamos que lograron resolver las 2000 peticiones y no se devolvió ningún TIMEOUT como lo hacía cuando sólo había una instancia del servidor tratando de cargar con todo el trabajo.

Los console.log demuestran cómo los diferentes procesos trabajan en conjunto

```
El worker 19232 ha atendido esta petición, el resultado es 124999999567108900
El worker 18596 ha atendido esta petición, el resultado es 124999999567108900
El worker 13988 ha atendido esta petición, el resultado es 124999999567108900
El worker 18308 ha atendido esta petición, el resultado es 124999999567108900
El worker 17660 ha atendido esta petición, el resultado es 124999999567108900
El worker 10212 ha atendido esta petición, el resultado es 124999999567108900
El worker 6756 ha atendido esta petición, el resultado es 124999999567108900
El worker 11564 ha atendido esta petición, el resultado es 124999999567108900
El worker 17268 ha atendido esta petición, el resultado es 124999999567108900
El worker 4392 ha atendido esta petición, el resultado es 124999999567108900
El worker 12636 ha atendido esta petición, el resultado es 124999999567108900
El worker 16872 ha atendido esta petición, el resultado es 124999999567108900
El worker 11460 ha atendido esta petición, el resultado es 124999999567108900
□
```

Para pensar

El ejemplo presentado en estas diapositivas fue realizado con un computador de 16 núcleos, lo cual significa que tuvimos un equipo de 16 servidores resolviendo el problema de las operaciones complejas.

Si tu computador tenía más o menos núcleos, ¿qué tanto cambió el resultado?



Estabilizador de workers

Duración: 5 – 10 minutos

CODERHOUSE



ACTIVIDAD EN CLASE

Estabilizador de workers

Ahora que comprendemos sobre los clusters

- ✓ Crear un servidor de express que levante **n** workers según sea el número de cpus() de tu computador.
- ✓ Confirmar en tu consola cuántos procesos con el nombre de imagen **node.exe** existen.
- ✓ Configurar los listeners del proceso primario para que, si alguno de sus workers muere en alguna operación o falla, el proceso primario cree una nueva instancia para siempre tener **n** número de workers estables.
- ✓ Hacer pruebas matando un worker con el comando **taskkill /pid PID -f** `taskkill /pid 14400 -f`
- ✓ Confirmar que el proceso primario haya creado el nuevo proceso y que el número de workers se mantenga estable.



Break

¡10 minutos y volvemos!

CODERHOUSE

Contenedores con Docker

CODERHOUSE

¡Atención!

Recuerda instalar `curl` para la próxima clase.

[¿Qué estoy por instalar?](#)



[Página oficial](#)

CODERHOUSE

Docker

Docker es una plataforma que permitirá crear, probar e implementar aplicativos en unidades de software estandarizadas llamadas **contenedores**.

Con docker, podremos “virtualizar” el sistema operativo de un servidor con el fin de realizar ejecuciones de aplicaciones con la máxima compatibilidad.

Gracias a tener nuestro aplicativo en un contenedor que corra un software con exactamente las especificaciones que necesita esta app, evitamos el típico problema del desarrollador “**en mi computadora sí funcionaba**”

- En mi computadora si funciona
- Si pero no le vamos a dar tu computadora al cliente



CODERHOUSE

¿Por qué mi aplicativo puede no funcionar al llegar al cliente?

El desarrollo de un aplicativo no es tan “ideal” como pensamos. Estamos hablando de múltiples desarrolladores haciendo código por su parte, para que al final “todo se una en un único código final.

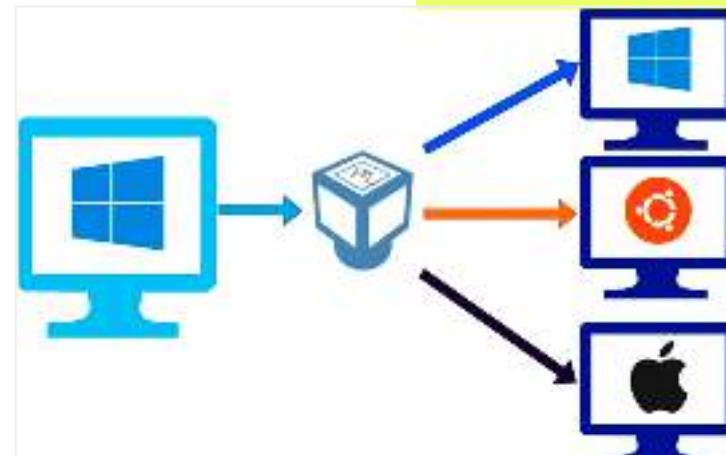
¿Cuáles podrían ser los puntos de dolor?

Cada computadora es un mundo, donde existe exactamente una versión específica de alguna librería, o exactamente una configuración particular del sistema, o exactamente una versión del entorno en general donde se ejecuta.

Necesitaríamos que todas las computadoras fueran exactamente iguales para asegurar una extrema compatibilidad en lo que ejecutamos.

¿Qué son las máquinas virtuales?

Como primera solución se plantearon las máquinas virtuales. **Grosso modo**, podemos decir que es cuando nuestra computadora utiliza sus recursos para **simular otro sistema operativo**. Eso significa que estamos “engañoando” a una computadora, haciendo creer que está corriendo sobre un entorno o sistema particular (y, en esencia, lo hace), sin embargo, todo esto en realidad está dentro de nuestra computadora principal, utilizando recursos de la computadora principal.



El problema de las máquinas virtuales



El cloud computing se basa en máquinas virtuales. Sin embargo, para desarrollo de aplicativos, realmente no parece ser la mejor opción contar con **todo un sistema operativo**, solo para una simple aplicación.

La idea debe estar entonces en ejecutar entornos que tengan **únicamente las configuraciones** necesarias para ejecutar una aplicación, y nada más. Esto es llamado **contenedor**.

Contenedor

Un contenedor es un entorno de ejecución para un aplicativo en particular, el cual tiene todas las dependencias que necesita dicha aplicación para poder correr sin problemas de compatibilidad.

La clave de un contenedor es el concepto del aislamiento, esto indicando que podemos tener múltiples contenedores, con diferentes entornos, con diferentes dependencias, y nunca habrá conflictos porque la instalación y uso de las dependencias se hace de manera interna.

Además, ya que el entorno no ocupa utilizar todo el sistema operativo (sólo el **kernel**), se vuelven realmente livianos en comparación con mover todo un sistema operativo en cada aplicativo.

El papel de Docker en el mundo de contenedores

Docker es una plataforma gestora de contenedores. Nos permitirá entonces empaquetar en un contenedor nuestro aplicativo, y posteriormente compartirlo a algún lado, para que al momento en el que tenga que ejecutarse, este pueda hacerlo dentro del contenedor aislado y asegurar que la ejecución será satisfactoria siempre.

La lógica de Docker se basa en tres pasos generales:

1

2

3

Paso 1

Un dockerfile:
Este cuenta con las
instrucciones paso a
paso para que
nuestro proyecto
genere una imagen.

Paso 2

Una imagen es el
equivalente de una **clase**,
pero con un proyecto
completo. Cuando
generamos la imagen de
una aplicación, significa
que podemos generar
múltiples contenedores a
partir de esa aplicación
(como **instancias**)

Paso 3

Contenedor:
El punto final en el
que ejecutamos el
aplicativo, pero esta
vez desde un entorno
cerrado.

Primer acercamiento a Docker: nuestra primera imagen

CODERHOUSE

1. Descarga de la página oficial

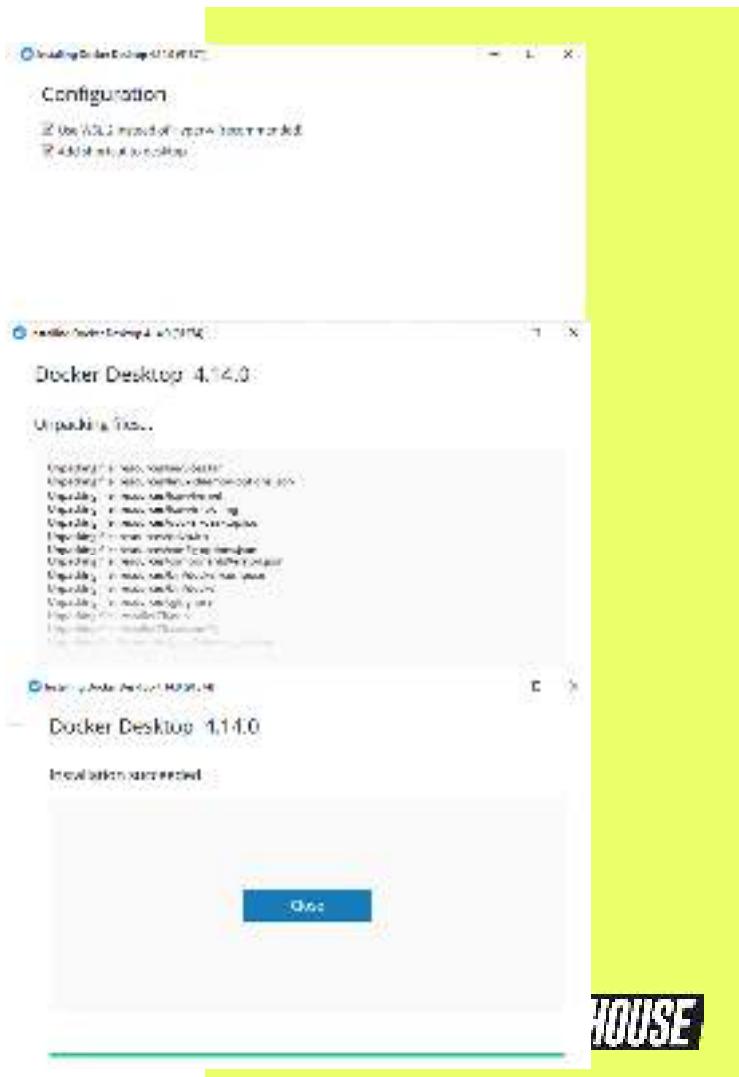
El paso más simple, vamos a descargar Docker Desktop a partir del sistema que necesitemos (En este caso la instalación será para Windows).



CODERHOUSE

2. Instalando

El instalador es bastante sencillo, al final Docker revisará que nuestro computador tenga **activada la opción de virtualización de Hardware**. Esta configuración se realiza desde el BIOS y es variable en cada computadora. Una vez que docker reconoce que podemos virtualizar, nos mostrará una pantalla como ésta:



HOUSE

¡Importante!

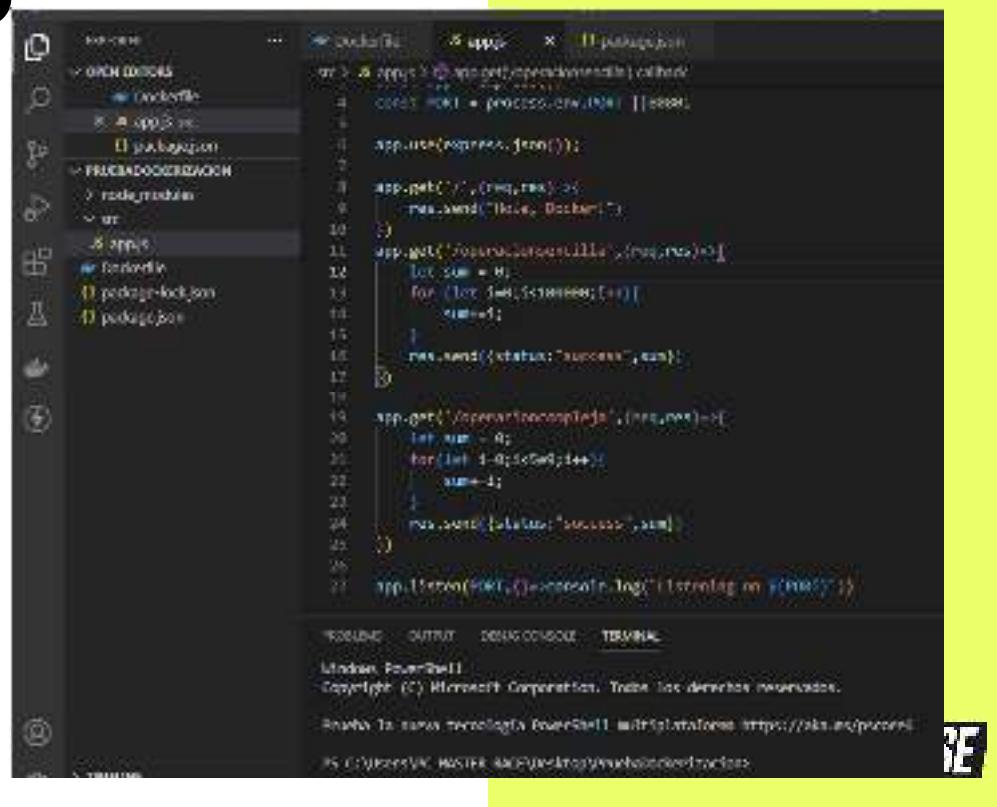
Si al momento de la instalación obtienes un error de Docker al querer inicializar los contenedores, significa que **necesitas activar la virtualización de hardware desde tu BIOS**.

Ya que es una configuración que requiere el reinicio del computador y entrar a la BIOS, es por ello que se solicitó su instalación previamente

3. Crear un Dockerfile en un proyecto

Tomaremos el proyecto que hace nuevamente la operación sencilla y la operación compleja, agregando el saludo de docker desde el endpoint de la ruta base. En dicho proyecto, crearemos un Dockerfile, que será el punto de partida de nuestra imagen.

CoderTip: También se recomienda contar con una extensión de Docker instalada en tu Visual Studio Code.



```
const express = require('express');
const app = express();
const port = 3001;

app.get('/', (req, res) => {
    res.send("Hello Docker!");
});

app.get('/operacionCompleja', (req, res) => {
    let sum = 0;
    for (let i = 0; i < 1000000000; i++) {
        sum += i;
    }
    res.send({status: "success", sum});
});

app.listen(port, () => console.log(`Listening on ${port}`));

```

The screenshot shows a Visual Studio Code interface with a dark theme. On the left is a file tree showing files like 'index.js', 'Dockerfile', 'package.json', and 'package-lock.json'. The main editor area contains the provided code for a Node.js application. Below the editor are tabs for 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The terminal tab shows some PowerShell commands related to Docker. A yellow bar at the bottom has the text 'IE' on it.

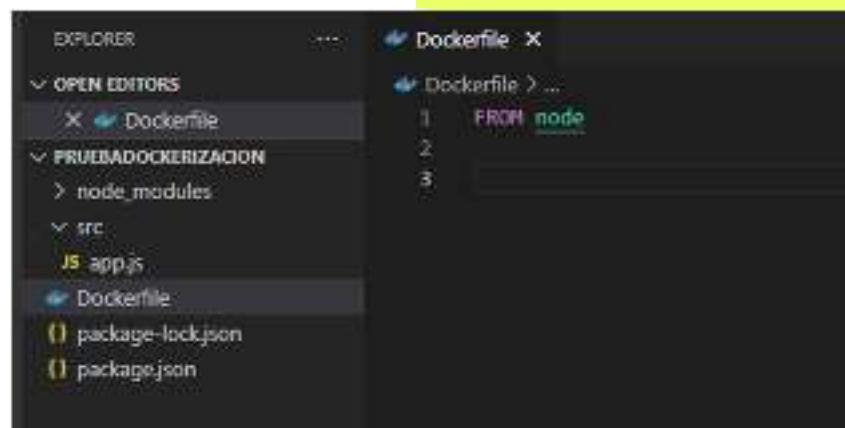
4. ¿Qué es una imagen base?

Escribimos en nuestro código:

FROM node., pero ¿qué significa?

Recuerda que una imagen es una plantilla para generar contenedores, es decir, hay muchas plantillas en las cuales podemos basarnos, para tener la configuración base de un entorno (que en este caso es node), y a partir de este comenzar a correr configuración más específica hacia nuestro proyecto.

FROM node, entonces, significa que estaremos tomand una imagen base del entorno de node, para poder configurar nuestra app.





Para pensar

Las imágenes base ya existen, deben tomarse de algún lado. ¿De dónde se descarga nuestra imagen para instalarse?

¡Existe un repositorio de imágenes!

Te invitamos a pasar por [DockerHub](#)

CODERHOUSE

5. Comenzamos a escribir en nuestro Dockerfile

Después del **FROM**, escribiremos el resto de la configuración:

WORKDIR Será nuestro directorio de trabajo principal, donde comenzaremos a crear todo. Las operaciones que hagamos más abajo se harán sobre este directorio

COPY permitirá copiar archivos de la carpeta donde estamos ejecutando el Dockerfile, y pegarlos en la carpeta que hayamos creado con **WORKDIR**



```
1 Dockerfile
2
3 # Nuestro directorio de trabajo principal
4 WORKDIR /app
5
6 # Creamos una carpeta dentro de /app para guardar nuestro proyecto (equivalentes a los
7 # directorios /src o /backend en otros frameworks)
8 COPY package*.json ./
9
10 # Instalación de dependencias necesarias para ejecutar el código
11 RUN npm install
12
13 # Ejecución del comando "npm start" que ejecuta el código
14 COPY . .
15
16 # Exponemos el puerto 3001 para que el navegador pueda acceder a la carpeta correspondiente
17 EXPOSE 3001
18
19 # Una vez finalizado, se deben ejecutar "npm start" para iniciar la aplicación (hay que ir al comando en tu package.json)
20 CMD ["npm", "start"]
```

5. Comenzamos a escribir en nuestro Dockerfile

RUN nos permitirá ejecutar comandos. Al usar la imagen base **node**, significa que el entorno podrá correr comandos de node y npm sin problema.

CMD al final es la ejecución del comando final que se utilizará al momento de echar a andar el servidor cuando hagamos **docker run**

```
# Dockerfile: X - Dockerfile
# Version: >
# Minimo definido: una versión mayor a 2020
# Hasta 2021
# Despues creare una carpeta llamada code para guardar nuestro proyecto (actualmente se ejecuta dentro de la carpeta /src)
# Versión: 2021
# Una vez ejecutado el comando docker build --tag tuImagen tuDockerfile se creara la carpeta /code con los siguientes archivos
# COPY package.json ./
# PROXY es usado el puerto 3001 por defecto y contiene un archivo Dockerfile que ejecuta el comando
# RUN rm -rf /var/www/html
# Despues de la ejecución, ejecutare el comando curl http://localhost:3001 para ver el resultado
# COPY .
# Esperando al usuario para que él sea quien elija el punto de ejecución deseado...
# PORT=3001
# Una vez realizado, se deben ejecutar "npm start" para iniciar la aplicación (ben luce el comando en tu package.json)
# CMD ["npm", "start"]
```

6. Ejecutamos el build

Una vez configurado nuestro respectivo dockerfile, podemos poner a prueba éste ejecutando el comando build.

El comando build leerá el archivo y comenzará con la construcción de la imagen para nuestro aplicativo.

Una vez que tenga la imagen del aplicativo, necesita colocarle un nombre, la flag **-t** significa "tag" y es para nombrar la imagen.

el punto **.** Sirve para indicarle que el **dockerfile** que necesitamos que lea está en la misma ubicación donde estamos corriendo el comando

```
docker build -t dockeroperations .
```

```
> [internal] load build definition from Dockerfile
> => transferring dockerfile: 32B
> [internal] load .dockerignore
> => transferring context: 2B
> [internal] load metadata for docker.io/library/node:latest
> [1/5] FROM docker.io/library/node@sha256:743787dbaca38ff4e3673a6e8c4d83d049e329de0f13e89e
> [internal] load build context
> => transferring context: 27.59kB
> [internal] CACHED [2/5] WORKDIR /app
> CACHED [3/5] COPY package*.json .
> CACHED [4/5] RUN npm install
> CACHED [5/5] COPY .
> => exporting to image
> => exporting layers
> => writing image sha256:a13f/e1a92efbdff0d43b55a2ab79d0932a5d20c6effec1d76ac0fb8534e6ce32c
> => naming to docker.io/library/dockeroperations
```

CODERHOUSE

¿Funcionó?

Para poder corroborar que el build se haya realizado correctamente, podemos ejecutar el comando `docker images`

Si ejecutamos el comando, deberíamos ver la imagen que construimos en la consola.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
dockeroperations	latest	a13f7eb192ef	18 hours ago	1GB

También podemos ver la imagen desde **Docker Desktop**, en la sección images

Solo nos queda hacer una cosa más: instanciar un contenedor a partir de esta imagen.



CODERHOUSE

Creando contenedores a partir de una imagen

CODERHOUSE

Creando contenedor desde CLI

La primera forma para poder crear un contenedor es desde un comando, podemos ejecutar:

```
docker run -p 8080:8080 dockeroperations
```

¿Por qué esa sintaxis del puerto? Recuerda que docker es un contenedor aislado, de manera que su “puerto interno 8080 (al que le hicimos EXPOSE)”, en realidad no existe en “el mundo real de nuestra computadora”. Entonces, necesitamos proveer un puerto para que realmente funcione en el exterior.

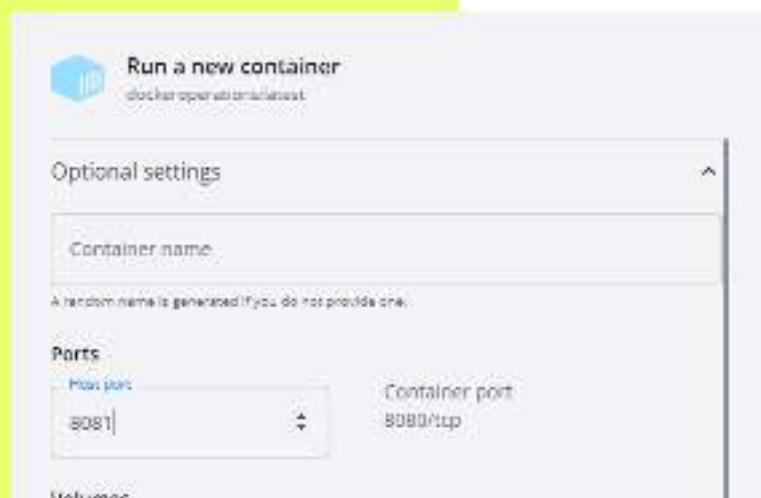
Es decir, de manera interna, el contenedor escucha en 8080, pero nosotros decidimos a qué puerto real lo conectamos.

Creando contenedor desde Docker Desktop

También podemos ir a Docker Desktop y seleccionar la opción “Run”



Creando contenedor desde Docker Desktop



Es importante tener conectado también un puerto de escucha al contenedor que estamos por crear, asignamos el 8081 (porque el otro lo utilizamos desde el CLI).

Tenemos dos contenedores, ejecutando el servidor

Cada servidor es independiente, pero ahora tenemos escuchando estos aplicativos en dos puertos: 8080 y 8081,

	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	agitated_euler 87cc25c9e03c ⏺	dockerooperations:latest	Running	8080:8080 ↗	19 minutes ago	
<input type="checkbox"/>	inspiring_keldysh 09aef12e82c3 ⏺	dockerooperations:latest	Running	8081:8080 ↗	8 minutes ago	

Los contenedores como instancias de la imagen

Recordando algo de Programación Orientada a Objetos (POO), sabemos que una clase permite generar múltiples objetos idénticos, pero con su propia identidad al final.

Cuando levantamos un contenedor, prácticamente estamos haciendo lo mismo. La imagen puede replicar múltiples veces el mismo proyecto, y así tener múltiples instancias del servidor.

¿Por qué es esto útil? Cuando hablamos de escalabilidad, tenemos que comenzar a pensar en múltiples procesos que puedan apoyarse entre sí, con el fin de poder atender un mayor número de peticiones. Con el modelo que recién creamos, si en algún momento necesitamos mayor potencia de procesamiento, podemos levantar un contenedor adicional para que se una al equipo.

Recuerda que esto es conocido como **escalamiento horizontal**.



¿Preguntas?

CODERHOUSE

¡Atención!

Recuerda instalar `curl` para la próxima clase.

[¿Qué estoy por instalar?](#)



[Página oficial](#)

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Clusterización
- ✓ Pruebas de performance en servidor clusterizado.
- ✓ Imágenes y contenedores con Docker

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 6. Testing y Escalabilidad Backend

Orquestación de contenedores

CODERHOUSE

Temario

5

Clusters & escalabilidad

- ✓ [Módulo Cluster](#)
- ✓ [Docker](#)
- ✓ [Docker como PM](#)

6

Orquestación de contenedores

- ✓ [DockerHub](#)
- ✓ [Orquestación de contenedores](#)
- ✓ [Orquestación con Kubernetes](#)

7

Seguridad

- ✓ [Cultura de seguridad](#)
- ✓ [OWASP](#)
- ✓ [OWASP Top 10](#)

Objetivos de la clase

- Subir imágenes de servidores a DockerHub
- Conocer Kubernetes
- Realizar un deploy de Kubernetes local con Minikube

CLASE N°5

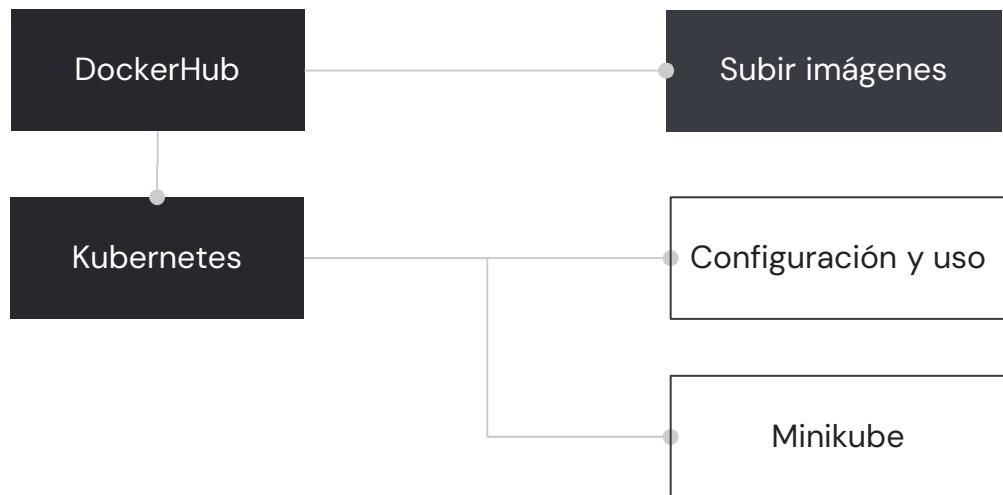
Glosario

Artillery: Es un toolkit de performance que prueba nuestro servidor y corrobora su fiabilidad en un entorno real.

Docker: es una plataforma que permitirá crear, probar e implementar aplicativos en unidades de software estandarizadas llamadas **contenedores**.



MAPA DE CONCEPTOS



Toda la clase es un proceso

CODERHOUSE

DockerHub

CODERHOUSE

¿Qué es DockerHub?

Dockerhub es una librería, o repositorio de imágenes en la nube.

Cuando hemos finalizado con el mantenimiento de nuestro aplicativo, necesitamos compartir la imagen resultante con nuestro equipo (Y enviar la imagen por correo electrónico no resulta óptimo).

Al subir nuestra imagen en la nube, todos los miembros autorizados podrán descargarla y utilizarla.



¿Por qué debería tener mis imágenes en la nube?

Existen múltiples razones para tener nuestra imagen en un repositorio:

- Podemos compartir nuestra imagen con nuestro equipo de desarrollo.
- Tenemos un sistema mejor controlado de nuestra imagen, ya que cada cambio de ésta puede significar una nueva tag con otra versión.
- Permite que otros softwares descarguen la imagen

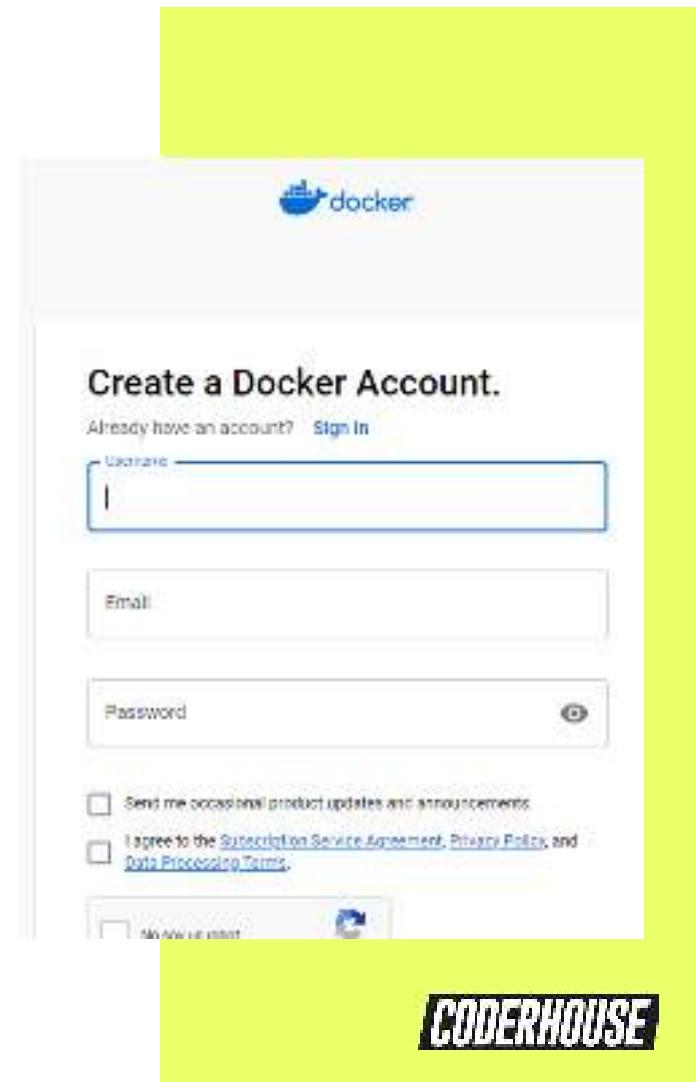
La razón de que éste se encuentre en la nube y permita que otros softwares lo descarguen, significa que al momento de configurar procesos más complejos (como deployment), podemos obtener la imagen directamente, sin necesidad de tener que estar cambiando el archivo local.

¡Disponible en todo momento y fácilmente actualizable!

Obtengamos nuestro Docker ID

Para comenzar a tener nuestras imágenes en nuestro repositorio, vamos a crear primero una cuenta.

Es importante recordar que el username que elijamos, se convertirá en nuestro Docker ID (lowercase)



The screenshot shows the 'Create a Docker Account' form on the Docker website. It features fields for Username, Email, and Password, along with two checkboxes for accepting terms and conditions. A 'Sign Up' button is at the bottom right.

Create a Docker Account.

Already have an account? [Sign In](#)

Username

Email

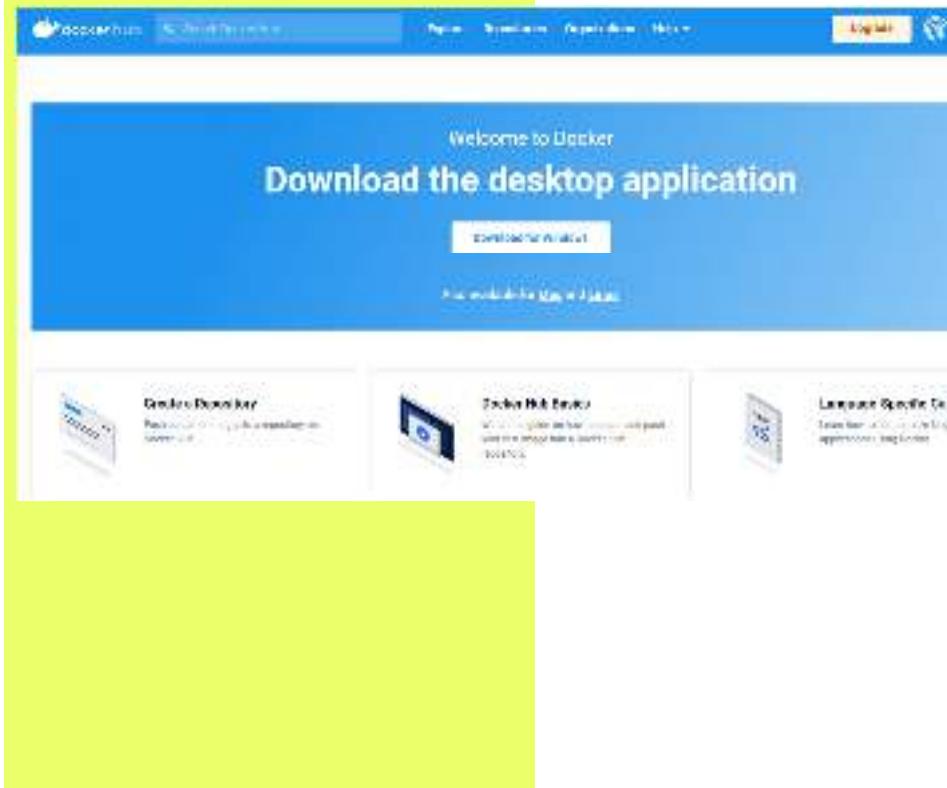
Password

Send me occasional product updates and announcements

I agree to the [Subscription Service Agreement](#), [Privacy Policy](#), and [Data Processing Terms](#).

CODERHOUSE

Panel principal



Una vez que confirmemos nuestra cuenta a partir del correo que recibimos, podemos entrar al panel principal de dockerhub

A partir de aquí, ya tenemos una cuenta activa para poder subir una imagen.

A pesar de que ya aprendimos el proceso para subir una imagen, nos falta aplicarlo a un proyecto más sólido, donde podamos aprovechar el potencial de la contenerización.

Subir una imagen a DockerHub

CODERHOUSE

 APROXIMACIÓN AL PROCESO

Comencemos con un proyecto para subir.

Con el fin de no afectar al flujo de desarrollo de un proyecto completo, el profesor cuenta con un proyecto que cuenta con las siguientes características

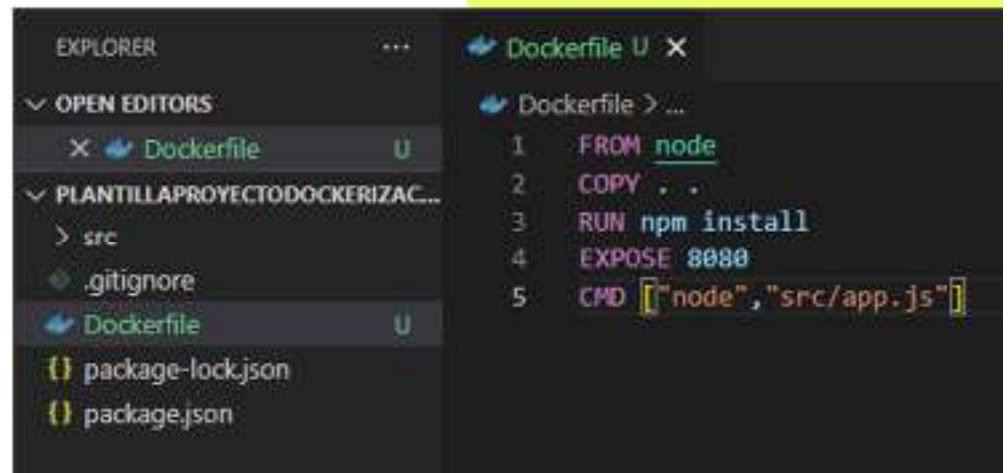
- ✓ Una vista para crear un usuario
- ✓ La creación de usuarios se hace vía mongoose, por lo cual se utiliza una variable de entorno MONGO_URL
- ✓ Además, se tiene una ruta test para generar usuarios (La cual se utilizará más adelante)

Se creará una imagen de docker para poder subir a DockerHub

APROXIMACIÓN AL PROCESO

Creando Dockerfile

Crearemos un Dockerfile con la configuración necesaria para tener una imagen ejecutable de nuestro proyecto.



The screenshot shows a code editor interface with two panes. The left pane is the Explorer view, which lists files and folders: 'Dockerfile' (marked with a green 'U'), 'src', '.gitignore', 'Dockerfile' (marked with a green 'U'), 'package-lock.json', and 'package.json'. The right pane is the code editor, showing the Dockerfile with the following content:

```
1 FROM node
2 COPY . .
3 RUN npm install
4 EXPOSE 8080
5 CMD ["node","src/app.js"]
```

APROXIMACIÓN AL PROCESO

Creando imagen

Ejecutamos el build de Docker y colocamos el nombre que deseemos (**no olvides el punto al final**)

```
docker build -t userscreator .
```

Deberá aparecer en nuestro dockerDesktop



CODERHOUSE

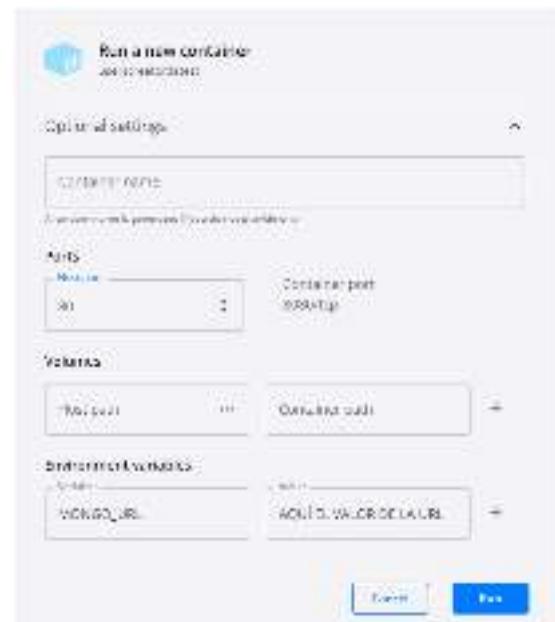
APROXIMACIÓN AL PROCESO

Levantemos un contenedor para probar la imagen

Vamos a ejecutar la imagen indicando el puerto donde deseamos conectar el puerto expuesto del contenedor.

Además, nota cómo colocamos la url de mongo para que el contenedor cuente con la variable necesaria para poder crear a los usuarios.

Debemos corroborar que el contenedor esté corriendo correctamente.



APROXIMACIÓN AL PROCESO

Preparamos la imagen para su subida: Login

Lo primero es conectar nuestro CLI a la cuenta de DockerHub que recién creamos. Para ello, llamaremos al comando

```
docker login |
```

Se nos solicitará el username (lowercase) y contraseña para poder acceder. Para corroborar que todo esté en orden, debemos llegar al siguiente mensaje:

```
Login Succeeded  
Logging in with your password grants your terminal complete access to your account.
```

CODERHOUSE

APROXIMACIÓN AL PROCESO

Preparamos la imagen para su subida: tagname

La estructura para cargar la imagen en dockerhub será:

<username>/imagen:version

Entonces, si nuestra imagen actualmente tiene el nombre **userscreator**. Bien podemos cambiar el nombre de la tag a partir de:

```
docker tag userscreator <username>/userscreator:1.0.0
```

Podremos ver la nueva imagen generada con el nombre indicado.

```
docker push <username>/userscreator:1.0.0
```

CODERHOUSE

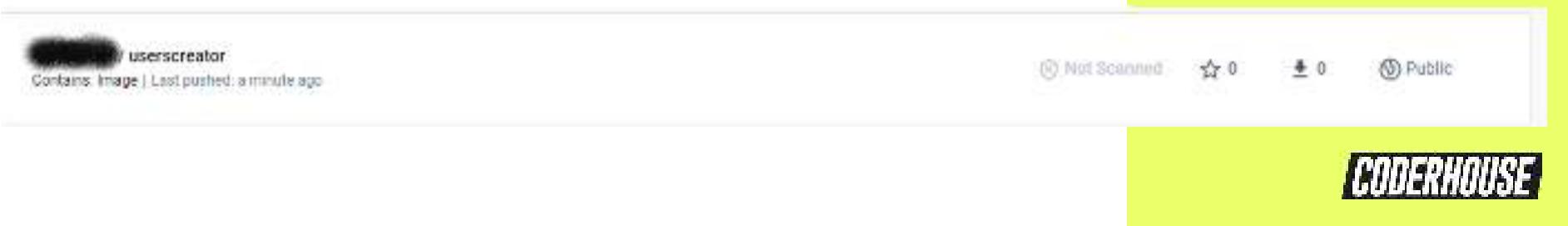
APROXIMACIÓN AL PROCESO

Finalmente, subimos la imagen a DockerHub

Muy similar a Github, para poder subir nuestra imagen ejecutaremos:

```
docker push <username>/userscreator:1.0.0
```

Ahora, en nuestro DockerHub, debemos poder acceder a nuestros repositorios y visualizar la imagen cargada. éste contará con la sintaxis **<username>/imagen** previamente mencionada.



Orquestación de contenedores

CODERHOUSE

Ya ganamos una batalla

Logramos ganarle a la sobrecarga de performance con la clusterización de Node. Además, conseguimos hacer nuestra primera virtualización de hardware, creando contenedores e incluso subiéndolos a la nube.

Sin embargo, queda mucho por resolver. Una de estas cosas es: si cada contenedor al final es un aplicativo corriendo de manera isolada, ¿cómo haremos para que todos los contenedores hagan armonía y se apoyen en las consultas de un único punto?

El entorno de producción es una guerra... ¡En verdad!

Cada vez nos hacemos más a la idea de que nuestro servidor no funciona “tan idealmente” como quisiéramos, cuando se trata de atender a un número realmente elevado de usuarios, con algunas peticiones de un nivel más complejo.

Es por ello que, si tenemos un elevado número de usuarios que consumen nuestros servicios, necesitamos responder con un número elevado de instancias para que no nos afecte en el rendimiento general.

¡Divide y conquistarás, hora de trabajar múltiples instancias de nuestra imagen, hora de ejecutar multi-contenedores!



PARA RECORDAR

Clusterización

Recuerda que con node podemos generar múltiples workers dirigidos por un Primary process. **Esta fue la primera alternativa de división de tareas que habíamos implementado.**

Lógica de clusterización para contenedores

Cuando hablamos de contenedores, muy seguramente escucharás acerca del término **Orquestación**. La orquestación es un término muy similar a la clusterización (a veces tomados por iguales), sin embargo, en esta clase haremos una ligera distinción.

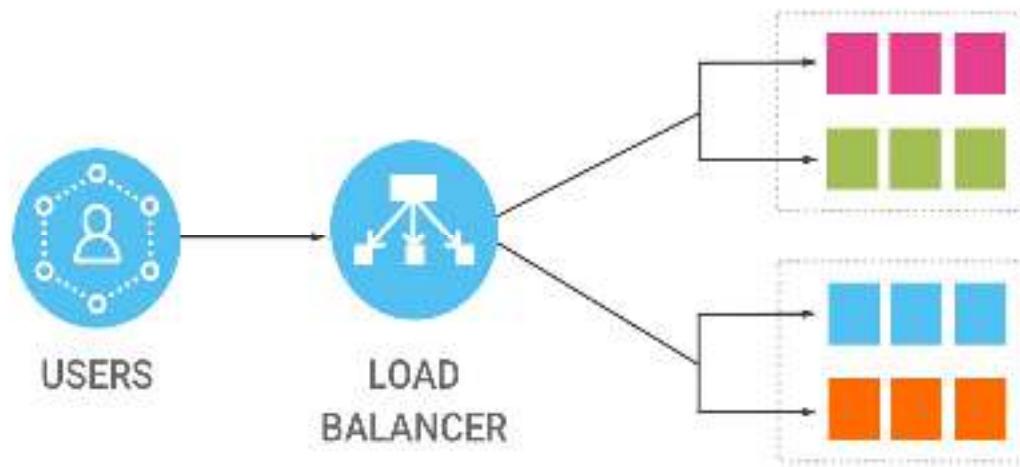
Al realizar una clusterización, estamos instanciando un modelo Primary-worker, en la cual los procesos workers pueden reiniciarse una vez caídos. ¿Y la orquestación? también se trata de tener un proceso principal, el cual se encargue de cargar con workers, sólo que ésta vez cada worker será en esencia un contenedor por sí solo.

¿En qué son diferentes?

La diferencia radica en que la orquestación es un proceso más profundo, ya que no sólo se trata de hacer la división, sino que permite realizar la gestión de éstos de manera más controlada, permitiendo:

- ✓ Control de reinicio para contenedores que presenten algún fallo
- ✓ **Balanceador de carga**, el cual permitirá que las peticiones puedan llegar a los contenedores de manera más distribuida, a diferencia de en cola (como lo hace la clusterización base).
- ✓ Manejo de rollout updates, los cuales nos permiten seguir un flujo organizado para poder actualizar o reconfigurar contenedores en un cluster

Balanceador de cargas





Break

¡10 minutos y volvemos!

CODERHOUSE

Orquestación con kubernetes

CODERHOUSE

¿Qué es Kubernetes?

Kubernetes es una tecnología de orquestamiento de contenedores. En esencia, se trata de una plataforma que sirve para administrar cargas de trabajos y servicios.

Kubernetes tomará un conjunto de instrucciones y las ejecutará para poder distribuir **pods**, donde cada pod puede tener **n** contenedores, de esta forma **todos los contenedores que pertenezcan a un pod, podrán funcionar como una entidad única para intercomunicarse**.

CODERHOUSE

¿Cómo comenzamos a utilizar Kubernetes?

Lo primero será comenzar a utilizar **kubectl**. El cliente de Kubernetes nos permitirá ejecutar comandos en nuestro CLI, de manera que lo necesitaremos instalado en nuestra computadora con Windows

Para poder instalar los binarios de kubectl directamente, ¿recuerdas que recientemente se te solicitó instalar curl?

Puedes hacer la instalación directamente corriendo este comando desde tu cmd:

```
curl.exe -LO "https://dl.k8s.io/release/v1.25.0/bin/windows/amd64/kubectl.exe"
```

Esto instalará directamente kubectl. Para poder probarlo, corre el comando ***kubectl version --short***

Para instalar en Linux, puedes revisar [aquí](#). Para Mac, puedes revisar [aquí](#)



¿Cómo obtener Kubernetes?

Kubernetes está pensado para poder ser desplegado, eso quiere decir que podemos hacerlo desde alguna plataforma en la nube, aunque en este caso, por cuestión de costos, lo utilizaremos de manera local.

Hay múltiples posibilidades para poder correr un repositorio local:

- ✓ Minikube
- ✓ MicroK8s
- ✓ k3s

En el desarrollo de esta clase, utilizaremos Minikube



minikube

CODERHOUSE

Minikube

Minikube es un software que permite levantar un cluster local de kubernetes, permitiendo hacer las pruebas que necesitemos dentro de nuestro contenedor de docker.

Funciona para cualquier sistema operativo y cuenta con un instalador directo.

- Para poder ejecutar el instalador, bastará con descargarlo de [este link](#). No hace falta gran configuración, todo se instalará directamente en la computadora.
- Para comenzar a trabajar con Minikube, bastará con correr el comando **minikube start**
- **Recuerda haber instalado previamente kubectl**, ya que al momento de iniciar minikube, buscará si está instalado, de otra forma lo tendremos que configurar manualmente.

```
Complementos habilitados: storage-provisioner, default-storageclass, dashboard
kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Corroborando las instalaciones

Una vez teniendo **kubectl** y **minikube**, vamos a poder correr el comando

```
• kubectl cluster-info
```

Podremos visualizar cómo tenemos Kubernetes corriendo en un puerto aleatorio: **Kubernetes control plane is running at https://127.0.0.1:54881**

Además, en nuestro docker desktop, podemos ver cómo contamos con la imagen de minikube:



Por último, ya deberíamos tener un contenedor de minikube corriendo



CODERHOUSE

Elementos importantes antes de comenzar

CODERHOUSE

Comandos utilizados comúnmente por kubectl

- ✓ **gets**: Son comandos utilizados para obtener información acerca de los recursos de kubernetes. Los gets más comunes realizados son los **kubectl get pods**, **kubectl get deployments** y **kubectl get services**
- ✓ **apply**: Nos servirá para poder crear nuestro primer recurso. El modelo de creación es declarativo, de manera que necesitaremos primero tener un conjunto de instrucciones que configuren el **pod** a crearse, además de los contenedores con los que se generará.
- ✓ **edit**: Servirá para cambiar la configuración de alguno de nuestros pods.

Comandos utilizados comúnmente por kubectl

- ✓ **delete**: Sirve para eliminar recursos, nuevamente, podemos realizar el borrado de pods y deployments completos.
- ✓ **config**: Sirve para comandos generales de configuración, por ejemplo, al activar kubernetes en nuestro docker desktop, reconocerá el contexto de nuestro kubectl automáticamente en minikube:

```
>kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	docker-desktop	docker-desktop	docker-desktop	
	minikube	minikube	minikube	default

Principales conceptos de Kubernetes

- ✓ **pod:** Unidad mínima de ejecución de kubernetes, éste consiste en un espacio donde podemos alojar nuestros contenedores. Podemos relacionarlos directamente con los **nodos** de un cluster. Si ejecutamos **kubectl get pods** obtenemos:

```
No resources found in default namespace.
```

- ✓ **replicas:** Refiere al número de pods a generarse, donde cada pod podrá contener una o n instancias de un contenedor.

Principales conceptos de Kubernetes

- ✓ **pod template:** Si sabemos que un pod contendrá uno o más contenedores, entonces necesitamos una plantilla para poder generar dichos contenedores, un template contendrá información sobre la replicación de los contenedores que deberá generarse una vez aplicada la configuración.
- ✓ **service:** Un recurso de servicio hace que los Pods sean accesibles para otros Pods o usuarios fuera del cluster. Sin un servicio, es imposible acceder a un pod. Gracias a éste, se podrá recibir un request y se redirigirá a los pods a partir de un balanceador de carga.

Principales conceptos de Kubernetes

- ✓ **deployment:** Es un creador y actualizador de **pods** y sets. A partir de un **manifiesto**, éste sabrá cómo crear el pod, qué contenedores colocar dentro de él, cuántas réplicas obtener de un contenedor
- ✓ **manifest:** El manifiesto es el archivo json o yml que cuenta con todas las instrucciones para realizar el deployment. Escribir correctamente un manifiesto llevará a un deployment exitoso y desembocará en la creación de nuestro primer pod.

Haciendo nuestro primer deploy en kubernetes

CODERHOUSE

Ya tenemos el cluster ejecutándose

Recuerda que podemos visualizarlo con el comando **kubectl cluster-info**.

Ahora, para poder levantar nuestros pods de kubernetes primero crearemos un **archivo de deployment**.

Éste contendrá toda la configuración de los pods a generar y los respectivos contenedores que tendrán dentro.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

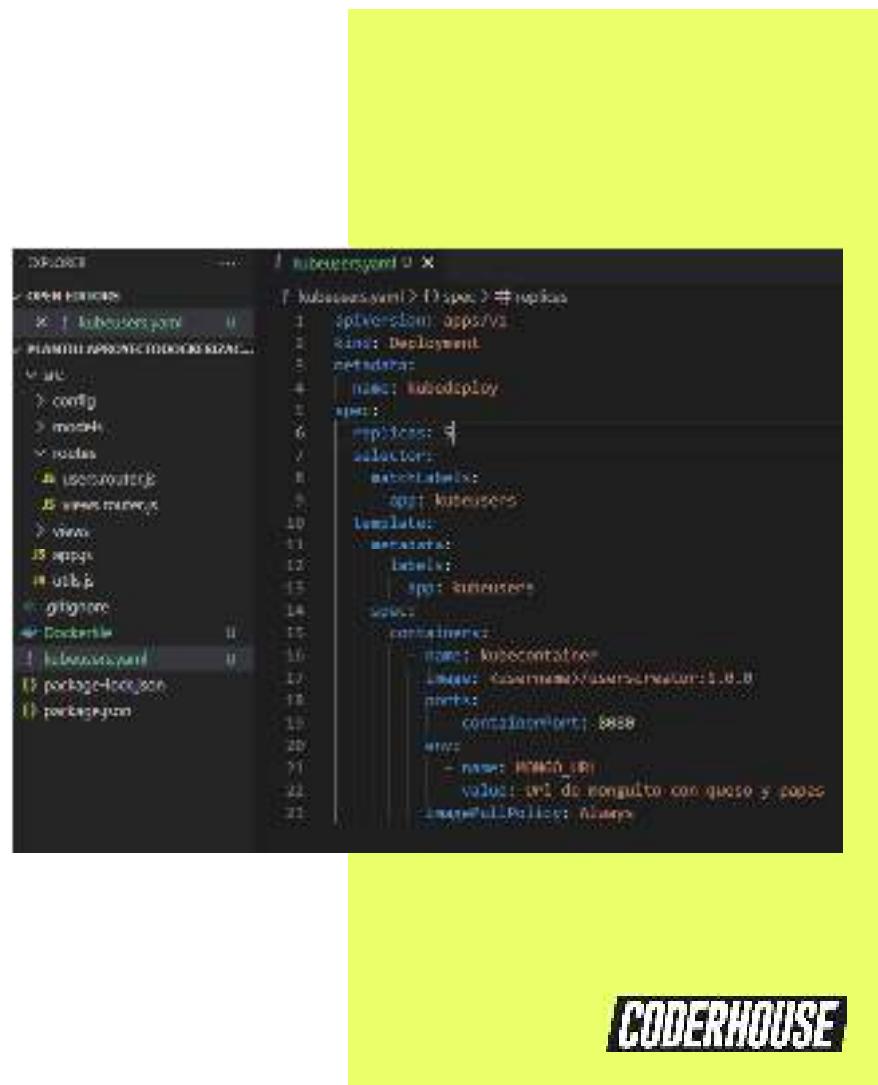
Ejemplo de un deploy de Nginx

CODERHOUSE

kubeUsers.yaml

Escribiremos entonces la configuración del cluster en un archivo con el nombre kubeusers (El nombre del app y labels debe ser en minúsculas), el cual tendrá el creador de usuarios que hemos trabajado.

Sin embargo, este archivo parece ligeramente más complejo que los que hemos visto, así que profundizaremos en las secciones



```
DEPRECATION WARNING: /etc/kubernetes/manifests/kubeusers.yaml is deprecated. Please use manifests/ instead.
$ kubectl get deployment kubeusers -n kube-system
NAME          READY   STATUS    RESTARTS   AGE
kubeusers     1/1     Running   0          10m
$ kubectl get svc kubeusers -n kube-system
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)   SELECTOR
kubeusers   <none>       <none>       8080/TCP  app: kubeusers
$ kubectl get pods -n kube-system
NAME          READY   STATUS    RESTARTS   AGE
kubeusers-0   1/1     Running   0          10m
$ curl -k https://192.168.1.11:8080
Welcome to the Kubernetes User API!
```

The screenshot shows a terminal window with several commands and their outputs. It starts by warning about the deprecated file path and then lists a Deployment named 'kubeusers' in the 'kube-system' namespace with one ready pod. It then lists a Service named 'kubeusers' with port 8080. Finally, it uses curl to access the service at https://192.168.1.11:8080, displaying the message 'Welcome to the Kubernetes User API!'. The terminal also shows the file structure of the kubeUsers.yaml file on the left.

Entendiendo un archivo de deploy: Base

- ✓ apiVersion: la versión del recurso que estamos trabajando:
- ✓ kind: El tipo de recurso, aquí hacemos saber que el recurso que estamos configurando es el deploy del aplicativo.
- ✓ metadata.name : El nombre con el cual será reconocida esta aplicación, este nombre aparecerá en cada pod como prefijo del pod

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubeusers
```

Entendiendo un archivo de deploy: spec

- ✓ spec: describe las especificaciones de lo que queremos que se implemente en este recurso.
- ✓ replicas: el número de pods que se generarán en el cluster de kubernetes.
- ✓ selector.matchLabels: hace referencia a que seleccione las instancias del template que cuenten con la label indicada. Podemos entonces leer: Genera 5 réplicas del template que cumpla con esta label.
- ✓ template: es la “plantilla” de nuestro pod, de manera que aquí describiremos los detalles de lo que contendrá dicho pod

```
spec:  
  replicas: 5  
  selector:  
    matchLabels:  
      app: kubeusers  
  template:  
    metadata:  
      labels:  
        app: kubeusers
```

CODERHOUSE

Entendiendo un archivo de deploy: template (1)

- ✓ spec (nuevamente): refiere a las especificaciones, pero esta vez hace referencia a las especificaciones de nuestro pod
- ✓ containers: Aquí definimos la característica del (o los contenedores) que vamos a meter en el pod
- ✓ name: Nombre del contenedor
- ✓ image: Imagen sobre la cual se basará el contenedor.
- ✓ ports.containerPort: hace referencia al puerto que el contenedor tenga expuesto

```
template:  
  metadata:  
    labels:  
      app: kubeusers  
  spec:  
    containers:  
      - name: kubecontainer  
        image: <username>/userscreator:1.0.0  
        ports:  
          - containerPort: 8080  
        env:  
          - name: MONGO_URL  
            value: Url de monguito con queso y papas  
        imagePullPolicy: Always
```

Entendiendo un archivo de deploy: template (2)

- ✓ env: variables de entorno, en este caso sólo tenemos configurada la variable de entorno MONGO_URL para nuestro proyecto.
- ✓ imagePullPolicy: sirve para determinar en qué momento debería obtener la imagen indicada de internet o del entorno local
 - IfNoPresent: Sólo hará pull de DockerHub si no está la imagen ya en el entorno local.
 - Always: Siempre pullear la imagen de DockerHub, independientemente de que se encuentre local.
 - Never: Nunca obtener de DockerHub, fuerza a que la imagen ya esté localmente (peligroso para minikube)

```
template:
  metadata:
    labels:
      app: kubeusers
  spec:
    containers:
      - name: kubecontainer
        image: <username>/userscreator:1.0.0
        ports:
          - containerPort: 8080
        env:
          - name: MONGO_URL
            value: Url de monguito con queso y papas
        imagePullPolicy: Always
```

```
! kubeusers.yaml U x
! kubeusers.yaml > {} spec > {} template :
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: kubeservice
5  spec:
6    selector:
7      app: kubeusers
8    ports:
9      - port: 80
10     targetPort: 8080
11   type: LoadBalancer
12 ---
13  apiVersion: apps/v1
14  kind: Deployment
15  metadata:
16    name: kubedeploy
17  spec:
18    replicas: 5
19    selector:
20      matchLabels:
21        app: kubeusers
```

El proceso aún no termina

Recuerda las definiciones importantes de kubernetes, comentamos que un servicio es aquel que realmente permitirá conectar con los pods de un cluster. Sin un servicio, nuestros pods quedarán ocultos en el cluster y nunca podremos realmente conectar un request con ellos.

El proceso aún no termina

```
! kubeusers.yaml U x
! kubeusers.yaml > {} spec > {} template :
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: kubeservice
5  spec:
6    selector:
7      app: kubeusers
8    ports:
9      - port: 80
10        targetPort: 8080
11    type: LoadBalancer
12  ---
13  apiVersion: apps/v1
14  kind: Deployment
15  metadata:
16    name: kubedeploy
17  spec:
18    replicas: 5
19    selector:
20      matchLabels:
21        app: kubeusers
```

Ahora escribiremos en el mismo archivo, el bloque superior indicado en la captura (nota cómo lo separamos del recurso de deploy).

Nota que el nuevo recurso es de tipo **Service**, además tiene el nombre **kubeservice**, y selecciona todos los pods kubeusers.

Por último, el tipo **LoadBalancer**, hace referencia a la distribución de requests entre los diferentes pods

¡Ha llegado el momento!

Toda nuestra configuración ahora tendrá sus frutos: Es hora de aplicar nuestra configuración para tener nuestro servidor en el cluster de kubernetes local de Minikube.

Para ello, colocaremos el comando:

```
kubectl apply -f kubeusers.yaml
```

Éste nos permitirá ejecutar el archivo yaml que estuvimos configurando. En cuanto lo ejecutemos, deberíamos poder ver:

```
service/kubeservice created  
deployment.apps/kubedeploy created
```

¡Vamos a revisar qué pasó!

Vamos a ejecutar un conjunto de comandos que permitirán revisar la integridad de lo que acabamos de deployar. A simple vista parece que no se ha ejecutado nada, sin embargo, hay que entender lo que se ejecutó

Primero está el ***kubectl get deployments***. El cual me ayudará a corroborar si el deploy se realizó satisfactoriamente:

<code>kubectl get deployments</code>	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
	kubedeploy	5/5	5	5	90s

¡Ahí está el nombre de nuestro deploy! además, indica que tenemos 5 pods “READY”. ¡Hay que ver dónde se localizan!

Visualizar pods

El siguiente comando será ***kubectl get pods***

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
kubedeploy-78f8d65d6b-6p84w	1/1	Running	0	3m17s
kubedeploy-78f8d65d6b-7hzl7	1/1	Running	0	3m17s
kubedeploy-78f8d65d6b-7js9c	1/1	Running	0	3m17s
kubedeploy-78f8d65d6b-kzd48	1/1	Running	0	3m17s
kubedeploy-78f8d65d6b-nkrgn	1/1	Running	0	3m17s

Podemos visualizar las 5 réplicas (pods) que se encuentran ahora ejecutándose (STATUS => Running).

Esto significa que estos 5 pods están corriendo un contenedor de Docker cada uno, lo cual me está permitiendo tener una fuerza mayor de procesamiento, además de estar controladas por un cluster y son tratados como una unidad.

Visualizar servicio (1)

El tercer comando a correr es **kubectl get services**. Éste nos devolverá el servicio que debimos haber levantado al hacer apply.

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	7d22h
kubeservice	LoadBalancer	10.97.148.45	<pending>	80:31599/TCP	6m21s

Ahí aparece el buen **kubeservice**, que es el que configuramos en nuestro .yaml

Recuerda que este servicio es el que permite conectar con los pods del cluster. Echa un vistazo a la columna PORT(s). ¿Notas cómo está haciendo el vínculo?

Significa que, el proceso que nosotros vinculamos a un puerto 80, está siendo procesado de manera interna en un puerto de escucha 31599.

Visualizar servicio (2)

Ahora, recordemos que al final nuestro cluster local al final vive en minikube, así que es momento de ponerlo a trabajar.

Primero, ejecutaremos **minikube service list**



NAMESPACE	NAME	TARGET PORT	IP
default	kubernetes	No node port	
default	kubesservice	80	http://192.168.49.2:31590
kube-system	kube-dns	No node port	
kubernetes-dashboard	dashboard-metrics-scraper	No node port	
kubernetes-dashboard	kubernetes-dashboard	No node port	

Uno de esos servicios debe tener el nombre del servicio que levantamos al aplicar el archivo de deploy. Ahora, para que pueda salir a la luz en una url de localhost, ejecutaremos **minikube service <nombre del servicio>**

Luego minikube buscará en su tabla el servicio solicitado y lo ejecutará para poder utilizarlo.

Ejecutando el servicio con minikube

The screenshot shows a terminal window with the following content:

```
minikube service kubeservice
```

Output:

NAMESPACE	NAME	TARGET PORT	URL
default	kubeservice	80	http://192.168.49.2:31599

Starting tunnel for service kubeservice.

NAMESPACE	NAME	TARGET PORT	URL
default	kubeservice		http://127.0.0.1:52786

At the bottom of the terminal window, there is a status bar with icons and the text "127.0.0.1:52786".

¡Bienvenido!, crea algunos usuarios

En conclusión...

La orquestación de contenedores y el acercamiento de configuraciones de nivel más complejo como son docker, kubectl, minikube y kubernetes, puede ser intimidante al principio. Sin embargo, es un stack de herramientas sumamente útil para que tengamos el primer acercamiento de una clusterización del servidor que hemos trabajado.

Estos procesos los irás aprendiendo sobre la marcha. ¡Ánimo y a profesionalizarlo a tu ritmo!



**Pre entrega de tu
Proyecto Final**

CODERHOUSE



ENTREGA DEL PROYECTO FINAL

Primera entrega

Se debe entregar

- ✓ Crear un router llamado mocks.router.js que funcione bajo la ruta base **/api/mocks**.
 - ✓ Mover el endpoint “/mockingpets” (Desarrollado en el primer Desafío Entregable) dentro de este router.
 - ✓ Crear un módulo de Mocking para generar usuarios de acuerdo a un parámetro numérico. Dichos usuarios generados deberán tener las siguientes características:
 - En “password” debe tener la contraseña “coder123” encriptada.
 - “role” puede variar entre “user” y “admin”.
 - “pets” debe ir como array vacío.
- ✓ Dentro del router mocks.router.js, utilizar este módulo en un endpoint GET llamado “/mockingusers”, y generar 50 usuarios con el mismo formato que entregaría una petición de Mongo.



ENTREGA DEL PROYECTO FINAL

Primera entrega

Se debe entregar

- ✓ Dentro del router mocks.router.js, desarrollar un endpoint POST llamado **/generateData** que reciba los parámetros numéricos “users” y “pets” para generar e insertar en la base de datos la cantidad de registros indicados.
- ✓ Comprobar dichos registros insertados mediante los servicios GET de users y pets

Formato

- ✓ Link al repositorio de Github con el proyecto completo, sin la carpeta de Node_modules.

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ DockerHub
- ✓ Orquestación de contenedores con Kubernetes
- ✓ Manejo de cluster local con Minikube

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser

- grabada

Clase 7. Testing y Escalabilidad Backend

Seguridad

CODERHOUSE

Temario

6

Orquestación de Contenedores

- ✓ DockerHub
- ✓ Orquestación de contenedores
- ✓ Orquestación con Kubernetes

7

Seguridad

- ✓ Cultura de seguridad
- ✓ OWASP
- ✓ OWASP Top 10

9

Documentación de API

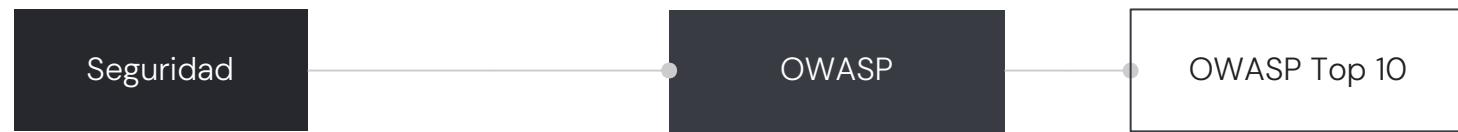
- ✓ Importancia de la documentación
- ✓ Documentar con Swagger

Objetivos de la clase

- Tener una noción sobre las vulnerabilidades de un sistema web
- Analizar el Owasp y resolver la teoría que plantea en entornos reales.



MAPA DE CONCEPTOS



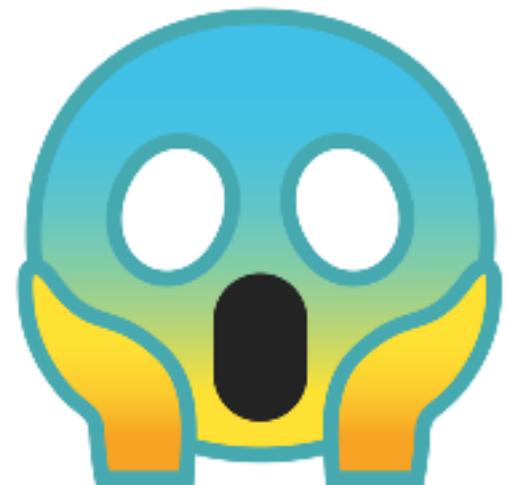
Cultura de seguridad

CODERHOUSE

El temor de todo desarrollador...

Si has llegado hasta este curso avanzado, seguramente tienes los conocimientos para desarrollar un servidor backend desde cero mediante buenas prácticas.

Sin embargo, hay algo más de lo que hay que hablar sobre el desarrollo, además de las arquitecturas, testing y escalabilidad:
las vulnerabilidades.



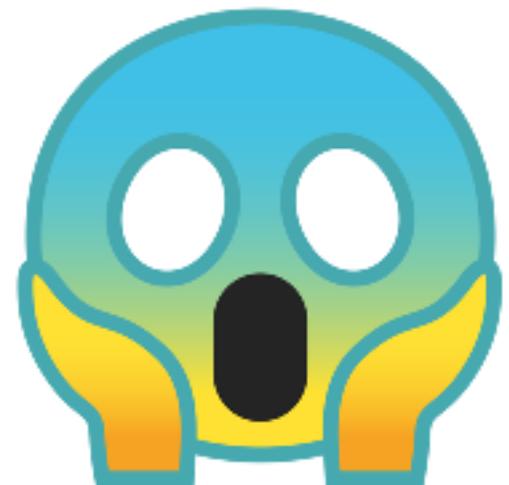
El temor de todo desarrollador...

Me imagino que en más de una ocasión habrás escuchado hablar sobre el hackeo de X plataforma, sobre el robo de información de X base de datos, o la simple caída de X página web debido a alguna persona malintencionada.

Cuando nuestro servidor se encuentre arriba, siempre tendremos este pequeño cosquilleo:

“¿Y si en algún momento, a causa de mi sistema web, alguien logra extraer información de mis usuarios?”

“No quiero que sea mi página web la que comprometa la información sensible de mis clientes”



CODERHOUSE

¿Qué es una vulnerabilidad?

En el mundo IT, una vulnerabilidad es cualquier tipo de debilidad o brecha dejada en nuestro sistema, que permita a una persona malintencionada aprovecharla para comprometer la seguridad del sistema o el usuario.



Sobre las vulnerabilidades

Existen múltiples tipos de vulnerabilidades que pueden comprometer a nuestra aplicación, éstas pueden ser:

- ✓ **De hardware:** cuando la implicación de seguridad se da en un elemento físico (como el servidor).
- ✓ **De software:** cuando la implicación de seguridad sí se da en nuestro aplicativo.
- ✓ **Procedimentales:** cuando la implicación de seguridad se da en el proceso mismo de resolución.
- ✓ **Humanas:** cuando la implicación de seguridad se da en una persona o usuario.

Sobre las vulnerabilidades

Cuando una persona malintencionada encuentra alguna de estas vulnerabilidades, puede tomar acción para **explotar** dicha vulnerabilidad y convertirlo en un evento bastante preocupante para nosotros.

¡Vamos a conocer algunas de las principales vulnerabilidades halladas en los sistemas web!

¡Importante!

El contenido de seguridad que estamos abarcando es desde un enfoque del Desarrollo Web.

Recuerda que el curso que estás tomando es de **backend**, no de **seguridad informática**. Para poder profundizar sobre la rama de informática, podrías hacer el curso de [Ciberseguridad](#).

OWASP

CODERHOUSE

¿Qué es OWASP?

¿Cómo podemos saber qué vulnerabilidades se están explotando en diferentes sistemas web del mundo?

Para eso, existe OWASP.

Open Web Application Security Project, un proyecto de código abierto **internacional**, sin fines de lucro, el cual brinda información referente a la seguridad general de aplicaciones web.

Además, OWASP provee un conjunto de herramientas que permiten conocer a profundidad muchos conceptos y herramientas referentes a la seguridad de aplicaciones web.



CODERHOUSE

¿Qué elementos podemos encontrar?

Owasp es muy amplio, y es mantenido por muchos elementos internacionales, lo cual nos permite acceder a herramientas como:

- ✓ [Owasp ZAP](#): Proxy para poder hacer testing de peticiones entre el navegador y aplicaciones que queremos probar.
- ✓ [Owasp Juice Shop](#): Aplicativo de prueba con vulnerabilidades, donde existen retos de búsqueda y explotación para detectar algunas de las vulnerabilidades de un proyecto "real".
- ✓ [Owasp Testing Guide](#): Guía sobre "Cómo testear tu aplicación web" para poder hacer un checklist de nuestra aplicación web, para verificar, vulnerabilidad por vulnerabilidad, si nuestra aplicación cuenta con ésta y podría significar algún futuro problema

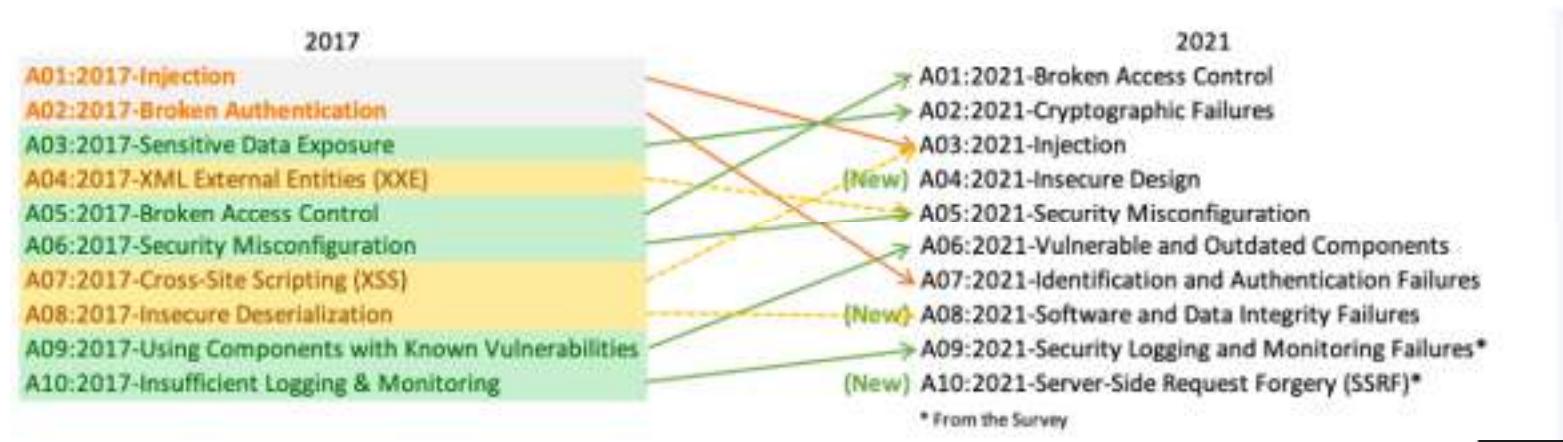
OWASP Top 10

CODERHOUSE

OWASP Top 10

Es un documento que muestra las 10 vulnerabilidades principales y de impacto más crítico en las aplicaciones web a lo largo de un determinado tiempo.

El top 10 es actualizado cada 3-4 años, con el fin de poder mantener actualizado el estado de las últimas prácticas que dejan expuestas a las aplicaciones web de los últimos años

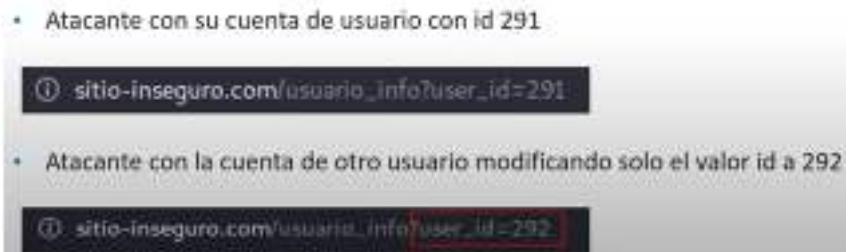


A01:2021 – Broken Access Control

Se encuentra cuando se logra acceder a un recurso al cual no debería tener acceso, permitiendo acceder a algún recurso o funcionalidad que va más allá de la autorización.

Aplica cuando:

- ✓ Se fuerza una URL para realizar búsquedas a otros recursos.
- ✓ Un usuario puede seguir navegando aún cuando su token de jwt haya expirado.
- ✓ Ruptura de una cookie con un token para poder elevar privilegios.



A02:2021 – Cryptographic failures

Se encuentra cuando se usa descuidadamente algún proceso que requiere un proceso criptográfico, ya sea que éste sea aplicado de manera errónea

Aplica cuando:

- ✓ Se utiliza algún algoritmo de cifrado obsoleto
- ✓ Cuando el proceso de hasheo tiene intervención directa del desarrollador (el cual en ocasiones puede no tener conocimientos sólidos de procesos de seguridad), lo cual genera una clave menos sólida de lo normal.

Cabe destacar que no siempre queda en nuestras manos. Si la librería externa que estamos utilizando no realiza un proceso de actualización correcto, podríamos también estar vulnerables a causa de un factor indirecto. (A06:2021 – Vulnerable and outdated components)

AO3:2021 - Injection

En esencia, se trata de cualquier inserción de información por parte del usuario con el fin de poder romper alguna consulta, o bien con el fin de obtener información sensible.

Aplica cuando:

- ✓ La data del usuario no fue validada correctamente (un req.body que se procesa directo sin desctructurarse o validarse).
- ✓ Un query dinámico sin validación contextual
- ✓ Una petición con parámetros malintencionados para poder realizar una consulta a la base de datos. (Inyección SQL o NoSQL por ejemplo)



A04:2021 - Insecure Design

No, no es CSS. El diseño se da en términos generales dentro de la construcción de una aplicación. Ésta no es una vulnerabilidad aislada, sino que son un conjunto de malas prácticas al momento de construir un aplicativo en general, desde pequeños detalles, hasta elementos estructurales más complejos.

Aplica cuando:

- ✓ ¿Has dejado un campo de password en texto plano, sin ocultar? ¿Cómo permitiste que la base agregara un producto con stock = -100 ?
- ✓ No se aplicaron patrones de diseño correctamente y las aplicaciones quedan fuertemente acopladas en funcionalidades.

Enter Password

CoderPassword210231

submit

A05:2021 – Security Misconfiguration

Se encuentra cuando tenemos algún descuido al momento de configurar el aplicativo, ya sea al configurar elementos internos del aplicativo, módulos o servicios de la nube

Sabemos que una aplicación no sólo puede fallar a causa del funcionamiento, sino también a causa de configuraciones de arranque, de algún servicio o de algún módulo.

A05:2021 – Security Misconfiguration

Aplica cuando:

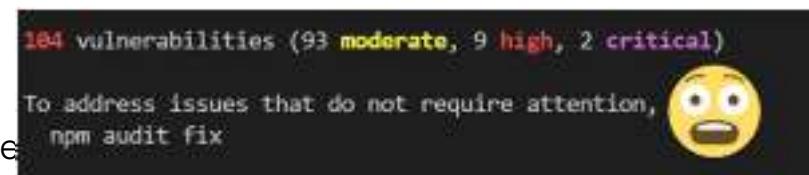
- ✓ Configuramos erróneamente los elementos de arranque del aplicativo (variables iniciales de argumentos o de entorno).
- ✓ Implementamos permisos o características **desmedidas** (Como setear el cluster de Mongo Atlas para recibir cualquier IP, 0.0.0.0/0)
- ✓ Configuramos erróneamente algún cloud service, generando inconsistencia o bloqueos en su uso (como configurar erróneamente un bucket de AWS S3, o setear mal una ruta de multer)
- ✓ Configuramos mal el manejo de errores y terminamos mostrando el stack trace a un usuario.

A06:2021 – Vulnerable and outdated components

Se encuentra al momento en el que un componente (ya sea externo o de nuestro aplicativo), se vuelve obsoleto o mantiene una vulnerabilidad que no ha sido subsanada en cierto tiempo.

Aplica cuando:

- ✓ No actualizamos librerías que tienen reportadas vulnerabilidades, haciendo posible que, si se sabe que nuestro aplicativo utiliza dicha dependencia, sea más fácil buscar la forma de explotarla.
- ✓ Dejamos módulos con documentación obsoleta o con features por implementar.
- ✓ Actualizamos un módulo sin probar si resolvía la vulnerabilidad inicial.



A07:2021 – Identification and authentication failures

Se encuentra a lo largo de todo el proceso en el que un usuario puede autenticarse. Éstas permiten que el usuario pueda forzar una autenticación, o acceder a información que haya sido tratada de manera errónea

Aplica cuando:

- ✓ Nuestro sistema de login permite rupturas de fuerza bruta (intentos automatizados de logueo)
- ✓ Guardamos contraseñas sin hashear
- ✓ Tenemos un sistema de **contraseña olvidada** erróneo, donde la aplicación prefiere **Recordarte la contraseña** en lugar de restablecerla directamente (Significa que la página tenía conocimiento de tu contraseña todo el tiempo).

```
email: "correodiego@correo.com"
psw: "123"
__v: 0
```

A08:2021 – Software and Data Integrity Failures

Se encuentra cuando hacemos un uso desmedido de módulos, librerías o integraciones generales de fuentes no confiables, o bien, que presentan alguna vulnerabilidad en algún momento.

Recuerda que, cada módulo que agregamos, significa hacer **dependiente a nuestra aplicación de lo que ocurra con dicho módulo.**

Si este módulo, esta integración, esta plataforma, llegara a tener algún problema, está fuera de nuestras manos el poder hacer algo al respecto, debido a la dependencia acoplada de ésto.

A08:2021 – Software and Data Integrity Failures

Aplica cuando:

- ✓ Alguna dependencia o integración presenta alguna falla, el exploit al final se hace por parte de esa herramienta, lo que nos deja vulnerables a nosotros también.
- ✓ Ocurre con dependencias que actualizamos sin revisar correctamente (¿Te acuerdas del caso [Aaron Swartz de Fakerjs](#)?)
- ✓ Ocurre con procesos CI/CD cuando la herramienta consigue ser vulnerada. (Caso serio de Heroku con Github a partir de la [vulnerabilidad de Travis-Cl](#))

A09:2021 - Logging and monitoring failures

Se presenta cuando no tenemos una correcta gestión de los logs del aplicativo y del monitoreo del mismo.

Aplica cuando:

- ✓ No tenemos un sistema de monitoreo de APIs para rastrear actividad sospechosa.
- ✓ Mostramos logs demasiado genéricos, que no dan información suficiente para rastrear y replicar un error.
- ✓ Mostramos logs demasiado explícitos, brindando el stack trace de manera general y no sólo cuando lo solicitemos.
- ✓ Nuestros logs sólo quedan de manera local y no hay forma de exportarlos para filtrarlos por prioridad.



A10:2021 – Server Side Request Forgery (SSRF)

Se encuentra cuando la aplicación no valida la URL que envía un usuario al momento de acceder a un recurso remoto.

Básicamente, el usuario malintencionado puede modificar la ruta a la cual se está accediendo al servidor, para intentar redireccionar a algún recurso al cual normalmente no estuviera pensado para acceder.

Aplica cuando:

- ✓ El servidor no tiene un control de redirecciones o restricción de URL por expresión regular.
- ✓ Cuando tenemos un sistema interno de callbacks, los cuales también pueden desembocar en redirecciones a recursos adicionales.

¿Qué hago con esta información?

Como se mencionó anteriormente, el mundo de la seguridad es mucho más amplio de lo que parece, y en muchas ocasiones va más allá del código mismo. El top 10 representa las principales vulnerabilidades de un aplicativo web, pero no es ni una décima parte del mundo de vulnerabilidades que hay en general en un entorno completo de una empresa.

¿Qué hago con esta información?

Aún con esto, por parte nuestra, podemos aportar siempre “un grano de arena”, cultivandonos sobre las buenas prácticas que nosotros podemos implementar en nuestros aplicativos, reduciendo así la posibilidad de un error a futuro.

Para practicar, cultivarse y mejorar tu expertise en seguridad de aplicaciones web, siempre puedes practicar con [todos los proyectos](#) que OWASP tiene de manera gratuita para ti.

CODERHOUSE



Break

¡10 minutos y volvemos!

CODERHOUSE

Maratón de detección de vulnerabilidades

CODERHOUSE

¡Es tu turno de ser el analista de vulnerabilidades!

Te enfrentarás a tres proyectos. Todos resuelven el mismo problema, sólo que irán aumentando ligeramente en implementaciones y dificultad sobre los cuales deberás hacer un análisis del código y de la implementación general para corroborar las diferentes vulnerabilidades que encuentres.

Para ello, encontrarás de manera individual las vulnerabilidades encontradas, después las clasificarás, y una vez pasado el tiempo, todo el grupo y el profesor podrán hablar acerca de las vulnerabilidades encontradas.



¿Cómo escalan los proyectos?

Proyecto 1

Ruteo, controladores,
registro de usuario

Proyecto 2

Ruteo, controladores,
registro de usuario,
bases de datos, login de
usuario

Proyecto 3

Ruteo, controladores,
registro de usuario, login
de usuario, vista de
perfil, manejo de sesión,
bases de datos, variables
de entorno, permisos.



#FindTheBug (1)

Encuentra el error

Analizaremos el código para hallar las vulnerabilidades del proyecto.



Clona el proyecto de aquí
(carpeta: proyecto1)

Duración: 10 minutos (5 individual + 5 grupal)

CODERHOUSE



#FindTheBug (2)

Encuentra el error

Analizaremos el código para hallar las vulnerabilidades del proyecto.



Clona el proyecto de aquí
(carpeta: proyecto2)

Duración: 15 minutos (10 individual + 5 grupal)

CODERHOUSE



#FindTheBug (3)

Encuentra el error

Analizaremos el código para hallar las vulnerabilidades del proyecto.



Clona el proyecto de aquí
(carpeta: proyecto3)

Duración: 15–20 minutos (10 individual + 5–10 grupal)

CODERHOUSE

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Seguridad
- ✓ Vulnerabilidades
- ✓ OWASP

Opina y valora
esta clase

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 8 . Testing y Escalabilidad Backend

Documentación de API

CODERHOUSE

Temario

7

Seguridad

- ✓ Cultura de seguridad
- ✓ OWASP
- ✓ OWASP Top 10

8

Documentación de API

- ✓ Importancia de la documentación
- ✓ Documentar con Swagger

9

Testing Unitario

- ✓ Módulos de testing
- ✓ Testing con Mocha
- ✓ Testing con Chai

Objetivos de la clase

- **Comprender** la importancia de un proceso de documentación
- **Conocer** Swagger como herramienta para documentar.
- **Aplicar** Swagger para documentar nuestros endpoints

CLASE N°7

Glosario

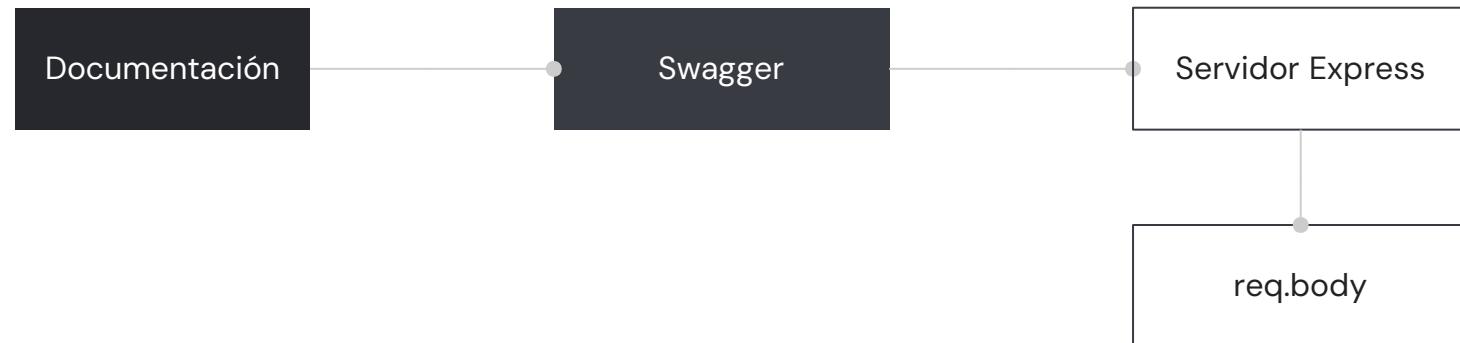
OWASP: Open Web Application Security Project. Es un proyecto de código abierto internacional, sin fines de lucro que brinda información referente a la seguridad general de aplicaciones web.

OWASP top 10: es un documento que muestra las 10 vulnerabilidades principales y de impacto más crítico en las aplicaciones web a lo largo de un determinado tiempo

Vulnerabilidad: cualquier tipo de debilidad en un sistema que permita ser aprovechada por una persona malintencionada para comprometer la seguridad del sistema o el usuario.



MAPA DE CONCEPTOS



Documentación

CODERHOUSE

Documentar

Documentar significa brindar suficiente información sobre algún proceso, con el fin de que éste sea lo suficientemente entendible para quien lo lea.

La documentación puede darse a nivel simple (comentarios sobre el código) o bien a nivel más complejo (herramienta de documentación para un aplicativo en general).



CODERHOUSE

¿Por qué es importante documentar?

Vamos a contextualizar algo bastante común en un ambiente laboral no controlado:

Estás trabajando en una empresa que se encarga de la venta de diferentes tipos de mueblería para el hogar, para esto, la empresa sólo procesa los muebles contra solicitud, de manera que manda a hacerlos a su fábrica productora sólo hasta que hayan solicitado una pieza con dichas características. Éste es todo el contexto que necesitamos.

Actualmente sólo trabajan dos desarrolladores en la empresa:

- ✓ Mauricio
- ✓ Tú

Cada quien está desarrollando diferentes módulos para la empresa: Mauricio se está encargando de la gestión de los muebles y el proceso de compra, y tú de usuarios y mantenimiento de CRM y CMS de la empresa.

Mauricio no tiene contexto de lo que estás armando, ni tú tienes contexto de lo que hace él.



¡Comienzan los problemas!

Por X cuestiones, Mauricio ha tomado la decisión de romper relaciones con la empresa, así que han decidido contratar a otra persona para apoyarte en el mantenimiento. ¿Cuál es el problema entonces? ¡Te has quedado a cargo de un código que no hiciste y, por lo tanto, del cual no tienes conocimiento!

Te solicitan entonces dar mantenimiento al primer módulo ajeno:

¿Confuso? ¡Espera a ver el código!

"Reajustar el proceso de gestión de órdenes de venta, para que esta vez permitamos agregar una pila de órdenes de compra premium, las cuales tendrán mayor prioridad que las órdenes de compra normales, y en consecuencia las órdenes de venta deberán primero abastecer a las órdenes premium antes que cubrir las órdenes normales. Sin embargo, si la orden de compra tiene más de 2 semanas desde su solicitud, hay que agregarla a las órdenes prioritarias. Actualmente, tienes la gestión de órdenes de compra normales, te toca agregar la consideración premium. ¿Para cuándo puedes entregarlo?"

CODERHOUSE

No puede ser tan malo, ¿verdad?

Lo primero que hay que hacer es entrar al código y entender el paso a paso del proceso que ya existe.

Al menos sabemos que tenemos que procesar órdenes de compra y órdenes de venta.

Así que buscas entre todo el código de la empresa, para poder llegar al código que esperabas encontrar, donde se procesa el cálculo de la repartición de órdenes de compra-venta (posteriormente compra/premium -venta)

¡Echemos un vistazo para poder entender el proceso!



Actividad colaborativa

Leemos el código de la distribución de órdenes, después y comentar que es lo que está haciendo el algoritmo y cómo puedes comenzar a modificarlo.

Duración: **10 minutos**

Función actual para distribuir órdenes de compra/venta

```
JS allocationAlgorithm.js X
JS allocationAlgorithm.js > ⚡ allocate
1
2  function allocate (salesOrders,purchaseOrders) {
3      if(!Array.isArray(salesOrders)||!Array.isArray(purchaseOrders)) throw new Error('Invalid data types. Both parameters must be strings');
4      const orderedSales = salesOrders.sort((a,b) => new Date(a.created) - new Date(b.created));
5      const orderedPurchases = purchaseOrders.sort((a,b) => new Date(a.receiving) - new Date(b.receiving));
6      const allocatedOrders = [];
7      let totalQuantityInStock = 0;
8      while(orderedSales.length>0&&orderedPurchases.length>0){
9          let currentPurchase = orderedPurchases.shift();
10         totalQuantityInStock+=currentPurchase.quantity;
11         while(totalQuantityInStock>=orderedSales[0].quantity)
12         {
13             const salesOrder = orderedSales.shift();
14             allocatedOrders.push({
15                 id:salesOrder.id,
16                 date:currentPurchase.receiving
17             })
18             totalQuantityInStock-=salesOrder.quantity;
19             if(orderedSales.length==0) break;
20         }
21     }
22     return allocatedOrders;
23 }
```

Nuestro trabajo no solo es entender ¡Falta modificar!

Hasta cierto punto, podemos decir que el código se explica por sí solo, ¿verdad?

Sin embargo, para poder comenzar a modificar este código, no basta con “hacerse a una idea de lo que hace”, sino que tenemos que entender a profundidad cómo lo resuelve, así podemos modificar código para agregar nuestra pila de órdenes premium, sin afectar tampoco la lógica principal del flujo.

Para realizar las modificaciones, comienzan a llegarernos muchas dudas.

- ✓ No hay comentarios que me puedan guiar.
- ✓ No hay ningún otro recurso que me apoye a saber cómo funciona o si hay algún punto en el que tenga que tener especial cuidado al momento de modificar.
- ✓ No hay algún ejemplo de input de salesOrders ni purchaseOrders, tendremos que buscarlo probando petición desde frontend (en caso de que se active desde front) o armar nuestro propio mock a partir de la base de datos.

En conclusión

Notarás cómo un código que no está correctamente documentado, básicamente es una bomba de tiempo que, si en algún momento necesita modificación, presentará múltiples complicaciones para el encargado de dicho mantenimiento.

El ejemplo anterior también puede aplicarse para:

- ✓ **Código de otras áreas de la empresa.**
- ✓ **Código propio que dejamos durante meses** (En ocasiones volver a un código que no hemos tocado en meses o años puede ser un infierno, aun cuando este sea nuestro).

Swagger

CODERHOUSE

¿Qué es Swagger?

Swagger es una herramienta de documentación de código, la cual nos permitirá mantener cada módulo de nuestra API dentro de un **espectro de entendimiento sólido**, es decir, todo se mantendrá en un contexto suficientemente alimentado de información, para poder ser entendido por futuros desarrolladores (O para una versión tuya del futuro), cuando tenga que revisar el código más adelante.

Con esta herramienta podremos hacer nuestra propia **Open API specification**



CODERHOUSE

Open API specification

También conocida como **Swagger specification**, es un formato de descripción de REST APIs.

Estas especificaciones pueden ser escritas en yaml o en json, y permitirán profundizar sobre un módulo, ruta o esquema específico de nuestra API

Por ejemplo, si queremos realizar la documentación de un módulo de usuarios ¿Qué habría que documentar?

Al desglosar el módulo, podríamos separarlo en la siguiente fórmula:

- ✓ Un esquema que represente al usuario.
- ✓ Un conjunto de rutas referentes a los usuarios.
 - ✓ Posibles queries para cada ruta
 - ✓ Parámetros para las rutas que sean necesarias.
 - ✓ Consideraciones especiales de cada endpoint
- ✓ Un conjunto de posibles Inputs para operaciones con el usuario.

APROXIMACIÓN AL PROCESO

Antes de comenzar, necesitamos un proyecto a trabajar.

Te presentamos **Adoptme**, un proyecto destinado a la adopción de mascotas, éste, en su versión inicial, cuenta con las siguientes características.

- Un sistema de usuario:
 - Obtiene todos los usuarios.
 - Obtiene un usuario a partir de su Id.
 - Crea un usuario con base en un método de registro.
 - Actualiza un usuario.
 - Elimina un usuario.
- Un sistema de mascotas:
 - Obtiene todas las mascotas.
 - Crea una mascota.
 - Actualiza una mascota.
 - Elimina una mascota.
- Un sistema de adopción:
 - Obtiene todas las adopciones.
 - Obtiene la adopción por su id.
 - Crea una adopción a partir de un usuario y una mascota.

 APROXIMACIÓN AL PROCESO

¿Qué haremos con este proyecto?

Este proyecto nos será útil para múltiples clases del futuro, de manera que te recomendamos tener el link para clonar el proyecto a la mano. No maneja entornos para mayor comodidad, así que solo tienes que reemplazar el link de mongo en el archivo app.js y ejecutarlo.

Puedes [clonar el proyecto aquí](#)

El objetivo es que en las clases donde utilicemos este proyecto, no tengamos que preocuparnos en armar las soluciones. En el peor de los casos solo tendremos que modificar algunas cosas.



Checkpoint: Revisión de código

Destina este tiempo para ejecutar el proyecto que acabas de clonar, da una leída a los endpoints y entiende el funcionamiento general. ¡Puedes comenzar a utilizarlo para sentirte más cómodo al documentarlo!

Tiempo estimado: **5 minutos**

CODERHOUSE

APROXIMACIÓN AL PROCESO

¡Hora de documentar! Instalación de Swagger

Lo primero será tener instalado Swagger, para ello, habrá que instalar dos dependencias:

- ✓ **swagger-jsdoc**: Nos permitirá escribir nuestro archivo .yaml o .json, y a partir de ahí generará un apidoc
- ✓ **swagger-ui-express**: Nos permitirá linkear una interfaz gráfica que represente la documentación a partir de una ruta de nuestro servidor de express.

```
npm install swagger-jsdoc swagger-ui-express
```

CODERHOUSE

APROXIMACIÓN AL PROCESO

Creamos las opciones principales de swagger

Desglosemos de qué se trata cada propiedad:

- ✓ **openapi:** Sirve para especificar las reglas específicas que seguirá la openapi generada.
- ✓ **title:** Título de la API que estamos documentando.
- ✓ **description:** Descripción de la API que estamos documentando.
- ✓ **apis:** Aquí especificamos la ruta a los archivos que contendrán la documentación. la sintaxis utilizada indica que utilizaremos una carpeta **docs**, la cual contendrá subcarpetas con cada módulo a documentar.

```
const swaggerOptions = {  
  definition:{  
    openapi:'3.0.1',  
    info:{  
      title:"Documentación del poder y del saber",  
      description:"API pensada para clase de Swagger"  
    },  
    apis:[`$__dirname__/docs/**/*.yaml`]  
}
```

APROXIMACIÓN AL PROCESO

Conectamos Swagger a nuestro servidor de Express

Con las opciones ya generadas, falta hacer la conexión final. tomaremos las opciones indicadas y colocaremos las siguientes líneas

```
const specs = swaggerJSDoc(swaggerOptions);
app.use('/apidocs', swaggerUiExpress.serve, swaggerUiExpress.setup(specs))
```

Ejecutemos el servidor y visitemos la ruta que recién definimos.



APROXIMACIÓN AL PROCESO

Así debería verse el archivo app.js en panorama completo:

```
DIRIGEN: └── app.js M X └── index.js
  └── node_modules
    ├── express@4.17.1
    ├── jsonwebtoken@0.3.2
    ├── mongoose@5.7.1
    ├── cookieParser@1.4.4
    ├── swaggerJade@1.0.0
    ├── swaggerUIExpress@3.0.0
    └── _stream_duplex@1.0.0
      └── index.js
    └── users
      └── router.js
    ├── petsRouter.js
    └── authRouter.js
  └── db
    └── connection.js
  └── services
    └── usersService.js
  └── utils
    └── logger.js
  └── index.js
  └── app.js
    └── swagger.js
  └── package-lock.json
  └── package.json
  └── .gitignore
```

```
1 import express from 'express';
2 import mongoose from 'mongoose';
3 import cookieParser from 'cookie-parser';
4 import swaggerJade from 'swagger-jade';
5 import swaggerUIExpress from 'swagger-ui-express';
6 import _stream_duplex from 'node:_stream_duplex';
7
8 import userRouter from './users/router';
9 import petRouter from './pets/pets.router';
10 import authRouter from './auth/auth.router';
11 import sessionRouter from './sessions/session.router';
12
13 const app = express();
14 const PORT = process.env.PORT || 3000;
15 const connection = mongoose.connect('mongodb://127.0.0.1:27017/test');
16 const swaggerOptions = {
17   definition: {
18     title: 'Documentación del poder y del caos',
19     description: 'API creada para clase de Swagger'
20   },
21   endpoint: 'https://$__dirname/docs/#!/yaml'
22 };
23
24 const spec = swaggerJade.swaggerOptions;
25 app.use('/api-docs', swaggerUIExpress(spec, swaggerUIExpress.customeRoutes));
26
27 app.use(express.json());
28 app.use(cookieParser());
29
30 app.use('/api/users', userRouter);
31 app.use('/api/pets', petRouter);
32 app.use('/api/auth', authRouter);
33 app.use('/api/sessions', sessionRouter);
34
35 app.listen(PORT, () => console.log(`Listening on ${PORT}`));
```



APROXIMACIÓN AL PROCESO

The screenshot shows a web browser window with the address bar displaying 'localhost:8080/apidocs/'. The main content area features a green header with the text 'Swagger' and 'supported by SMARTBEAR'. Below this, a large heading reads 'Documentación del poder y del saber'. A subtext below the heading says 'API pensada para clase de Swagger'. At the bottom of the page, the message 'No operations defined in spec!' is displayed. The background of the slide has a repeating watermark pattern of the words 'CODERHOUSE'.

localhost:8080/apidocs/

Swagger
supported by SMARTBEAR

Documentación del poder y del saber

API pensada para clase de Swagger

No operations defined in spec!



Break

¡10 minutos y volvemos!

CODERHOUSE

Escribiendo nuestro archivo de configuración

CODERHOUSE

Swagger reportándose al servicio

Una vez llegando a la vista principal, podremos visualizar una interfaz gráfica lista para comenzar a mostrar los elementos que queramos documentar.

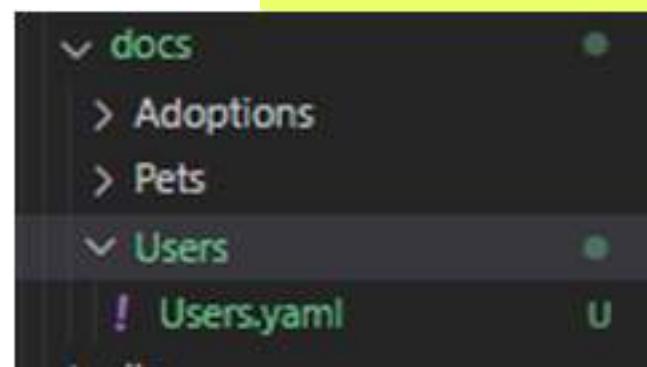
El resto consiste en escribir nuestros .yaml para comenzar con las especificaciones. Una documentación como mínimo debe contar con los siguientes elementos:

- ✓ Schemas.
- ✓ Routes.
- ✓ Inputs.
- ✓ Responses.

Con estos elementos podemos comenzar a estructurar una documentación sólida para un proyecto.

Crearemos una carpeta para cada respectiva entidad

Según los elementos que haya que documentar, cada entidad tendrá una carpeta con un archivo correspondiente a dicha entidad. Sobre este archivo escribiremos todos los elementos que enlistamos anteriormente.



Definiendo rutas

La palabra reservada **paths** sirve para colocar cada ruta que se encuentre en nuestro respectivo router. Ésta contendrá, de manera indentada, todos los métodos que estén relacionados con esa ruta (en este proyecto sólo está relacionado con el método get)

Cada método puede tener una breve descripción de la intención del endpoint, así también como una etiqueta para agrupar en la documentación.

Una vez escrita, podemos visualizar los cambios directamente en la interfaz gráfica que tenemos de Swagger

```
! Users.yaml U ×
RecursosBackend-Adoptme > src > docs > Users > ! Users.yaml > ...
1 paths:
2   /api/users/:
3     get:
4       summary: Obtiene todos los usuarios
5       tags:
6         - Users
```

Documentación del poder y del saber



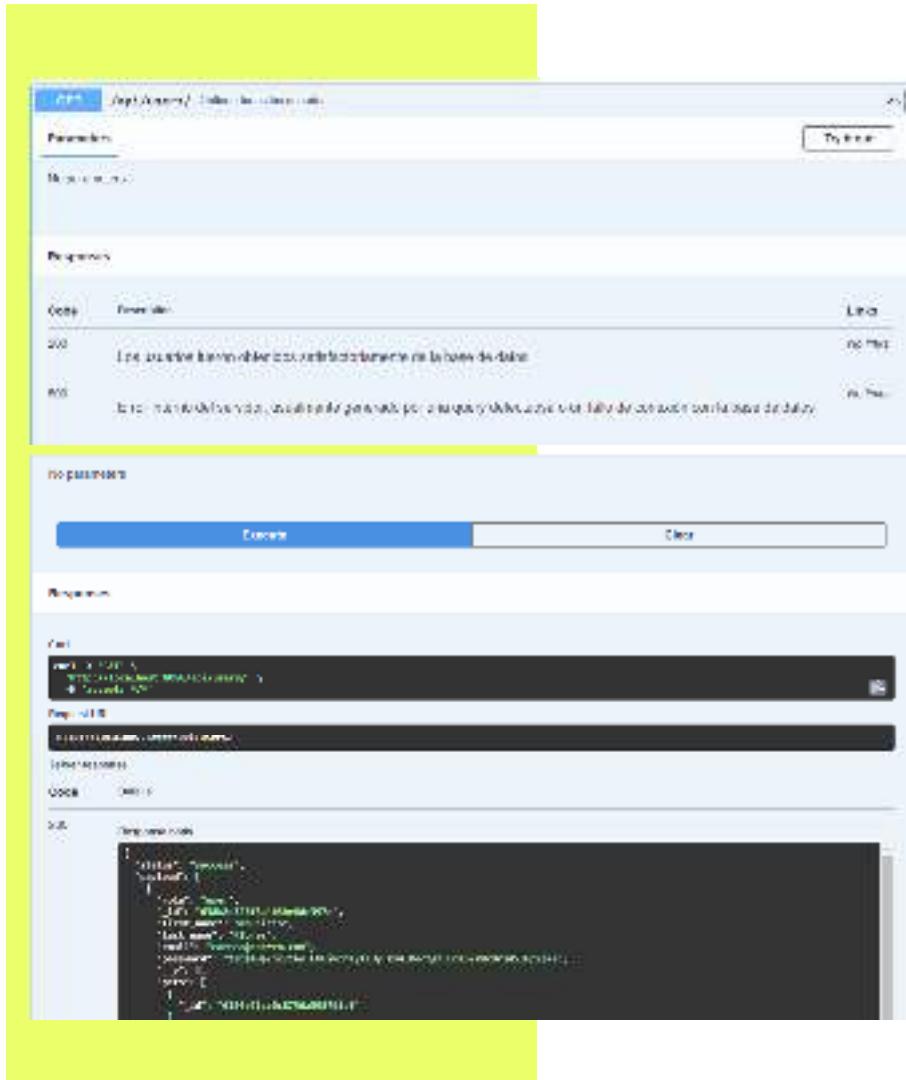
```
get:
  summary: Obtiene todos los usuarios
  tags:
    - Usuarios
  responses:
    "200":
      description: Los usuarios fueron obtenidos satisfactoriamente de la base de datos
    "500":
      description: Fue un error del servidor, usuario generado por un query incorrecto o un fallo de conexión con la base de datos
```

GET /api/users/:id/obtener-los-usuarios		
Parámetros	Try it out	
No parámetros		
Responses		
Code	Description	Links
200	Los usuarios fueron obtenidos satisfactoriamente de la base de datos.	View
500	Error interno del servidor, usualmente generado por una query defectuosa o un fallo de conexión con la base de datos.	View

Respuestas

Cada método puede devolver diferentes statuses al momento de procesarse, de modo que es bueno informar qué representa cada uno de estos errores, para ser más informativo.

Podemos notar cómo, si abrimos el endpoint generado por Swagger, vienen dichas descripciones ligadas al método y al status.



Try it out

¿Notas el botón en la esquina superior derecha que dice "Try it out"? ¡Maravilloso! Swagger no sólo se encarga de informar sobre el endpoint, sino que también nos da la posibilidad de ejecutar dicha consulta al alcance de un botón.

Al presionar el botón, obtendremos más abajo información de la petición que se realizó, además del resultado obtenido.

Generando componentes

Sería maravilloso contar con un ejemplo de un usuario, es decir, en múltiples ocasiones tenemos problemas para abstraer y proyectar un objeto, utilizando Swagger podemos generar un **schema** de usuario, para tener una idea más clara de **qué es lo que realmente representa esa entidad en nuestro aplicativo.**

Ahora, debajo de **paths** en nuestro archivo users.yaml, procederemos a definir otro elemento llamado **components**.

Los componentes pueden contener:

- ✓ Esquemas de una entidad.
- ✓ Modelos de respuestas.
- ✓ Esquemas de Inputs para un método particular

Definiendo un schema de tipo User. Nota cómo cada propiedad de este esquema cuenta con un tipo de dato y una descripción de lo que representa.

Además, un ejemplo de cómo debería verse un objeto en la redacta.

```
File: users.js X 8 app.js M
ResourceBackend-Simpli 3 nc 3 docs 3 Users 3 [ ] Users.js 3 () components 3 () schemas 3 () User 3 () example 3 () password
  + 3897:
    description: Fehler interno del servidor, usualmente generado por una query defectuosa o un fallo de conexión con la base de datos.
  12 COMPONENTS:
  13   schemas:
  14     users:
  15       type: object
  16       properties:
  17         _id:
  18           type: ObjectId
  19           description: Id autogenerado de mongo
  20         first_name:
  21           type: String
  22           description: Nombre del usuario
  23         last_name:
  24           type: String
  25           description: Apellido del usuario
  26         email:
  27           type: String
  28           description: Correo del usuario, este campo es único
  29         password:
  30           type: String
  31           description: Contraseña hashizada del usuario.
  32       example:
  33         _id: ObjectId("5eab23bf3a4bb6edc397e")
  34         first_name: Mauricio
  35         last_name: Espinoza
  36         email: correo@correo.com
  37         password: $2b$08$2jz7zGgTEOF1kbhKyIIly.kx4T0ctgt30mtu/WelUnoSTDp2jk
```

Swagger reconoce el schema, muestra los campos bien explicados, además de un ejemplo de cómo debería verse un valor real.

Schemas

User ↗ {

_id	ObjectId	Id autogenerado de mongo
first_name	String	Nombre del usuario
last_name	String	Apellido del usuario
email	String	Correo del usuario, este campo es único
password	String	Contraseña Hashed del usuario.

}

example: OrderedMap { "_id": "ObjectId('538b8c32f3abb3be0dc397e1')", "first_name": "Mauricio", "last_name": "Espinoza", "email": "correo@correo.com", "password": "\$2b\$10\$7jc7z6oTEGF1nbHRYI1Uy.nh4T0ctgt30dzu/WlxUr5dSTDp2jka" }

Referenciando esquemas en nuestro documento

Ahora, tenemos un esquema ya definido en nuestros componentes, ¿cómo hacer para indicar que el método get en `api/users/` devuelva un array de usuarios?

Para esto, podemos indicar en nuestro respectivo status un valor **content**, el cual contendrá el cuerpo de ejemplo de la respuesta, usaremos un **\$ref**, con el fin de referenciar a algún componente más del documento.

Revisa cómo se quedó la documentación una vez añadido el schema

```
responses:
  "200":
    description: Los usuarios fueron obtenidos satisfactoriamente de la base de datos
    content:
      application/json:
        schema:
          type: array
          items:
            $ref: "#components/schemas/User"
  "500":
    description: Error interno del servidor, usualmente generado por una query defectuosa
```

Parametrizando

Notarás que, en su mayoría, los endpoints del router de usuarios tienen un :uid, haciendo referencia a la id del usuario. ¿Cómo se agrega ésto a la documentación? Para ello, usaremos la sintaxis {param}

Una vez indicado, Swagger habilita un campo para poder pasarlo como parámetro



```
components/schemas/User
  "500":
    description: Error interno del servidor, usualmente p
  /api/users/{uid}:
    get:
      summary: Obtiene un solo usuario a partir de su ID
      tags:
        - Users
      parameters:
        - name: uid
          in: path
          required: true
          description: id del usuario que se desea solicitar
          schema:
            type: String
      responses:
        "200":
          description: Usuario encontrado
          content:
            application/json:
              schema:
                $ref: '#components/schemas/User'
```

CODERHOUSE

Definiendo el req.body

Pensemos en nuestro req.body como un input o un **requestBody** como gustan llamarlo en Swagger. Al final, lo primero es tener este Input inicial separado en un componente nuevo.

En nuestro mismo archivo, debajo del componente “schemas”, hay que crear otro componente llamado `requestBodies`, sobre el cual guardaremos un input “`updateUser`”.

updateUser entonces representará el usuario que llega por req.body, para poder hacer la actualización.

```
        email: correo@correo.com,
        password: $2a$08$cjC2eoVc0PohkeyIly.ks4l0octgtb0zu/M4tun051bpqfka
      },
      roles: [
        'admin',
        'user'
      ],
      properties: {
        first_name: {
          type: String,
          description: Nombre del usuario
        },
        last_name: {
          type: String,
          description: Apellido del usuario
        },
        email: {
          type: String,
          description: Correo del usuario, este campo es obligatorio
        },
        password: {
          type: String,
          description: Contraseña del usuario, posteriormente será hasheados
        }
      },
      examples: [
        {
          first_name: 'Karlos',
          last_name: 'Gutiérrez',
          email: 'correo@correo.com',
          password: '123'
        }
      ]
    }
  }
}
```

CODERHOUSE

Uso de req.body

Una vez que ya está definido el req.body, sólo basta con referenciarlo en el endpoint **put** que armaremos en el mismo path **api/users/{uid}**

Recuerda que, debido a que sólo es un método diferente al **get**, pero sigue siendo la misma ruta, hay que colocar el método put al nivel de get para que la documentación lo reciba correctamente.

```
openapi: 3.0.1
  ...
  paths:
    /api/users/{uid}:
      put:
        summary: Actualiza un usuario a partir de su id
        tags:
          - Users
        parameters:
          - name: uid
            in: path
            required: true
            description: id del usuario que se desea solicitar.
        schema:
          $type: String
        requestBody:
          required: true
          content:
            application/json:
              schema:
                $ref: '#components/requestBodies/updateUser'
```

Prueba de req.body

Nota algo interesante, cuando definimos un body y presionamos **try it out**, nota cómo en el cuadro de abajo se genera automáticamente un json de ejemplo listo para enviar.

Esto permite entonces que no nos perdamos en la idea de “¿Cómo probar un servicio en particular?” Ya que no tenemos que buscar el input de prueba que corresponde, pues Swagger ya nos genera el objeto de prueba automáticamente, y sólo necesitamos cambiar los campos del body



Importante

Seguramente estarás pensando: "Se hace bastante robusto ese archivo!" y querrás separar la lógica de documentación en más sub-archivos, sin embargo, la lógica de múltiples archivos actualmente se encuentra inestable por parte del repositorio principal, lo que está presentando múltiples errores.

¡Se recomienda que sigas trabajando con un archivo por entidad!



Hands on lab

En esta instancia de la clase **repasaremos** algunos de los conceptos vistos en clase con una aplicación

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **30 minutos**

CODERHOUSE



HANDS ON LAB

Proceso de adopción desde Swagger

¿Cómo lo hacemos? **Se crearán los endpoints correspondientes en la documentación para llevar a cabo un proceso de adopción**

- ✓ Definir la documentación para mascotas, sólo es necesario definir el método get y el post para crear y ver las mascotas.
- ✓ Definir la documentación para procesar el método “register” que se encargará de crear un usuario. No es necesario implementar el login, sólo el registro.
- ✓ Definir la documentación de adopción, la cual deberá recibir doble parámetro para poder llevar a cabo el proceso de adopción.
- ✓ Corroborar en la base de datos que las entidades se estén creando correctamente.



Documentar API

CODERHOUSE



ACTIVIDAD PRÁCTICA

Documentar API

Consigna

- ✓ Realizar la configuración necesaria para tener documentado tu [proyecto principal](#) a partir de Swagger.

Aspectos a incluir

- ✓ Se debe tener documentado los módulos de:
 - Sessions.
 - Pets.
 - Adoptions.

Formato

- ✓ Link al repositorio de Github sin node_modules

Sugerencias

- ✓ Recuerda que es un proceso de documentación, ¡Hay que ser lo más claros posibles!

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Importancia de la documentación
- ✓ Swagger
- ✓ Manejo de CRUD documentado con Swagger

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Esta clase va a ser
• grabada

CODERHOUSE

Clase 9. Testing y Escalabilidad Backend

Testing unitario

Temario

8

Documentación

- ✓ Importancia de la documentación
- ✓ Documentar con Swagger

9

Testing Unitario

- ✓ Módulos de testing
- ✓ Testing con Mocha
- ✓ Testing con Chai

10

Testing Avanzado

- ✓ Tests de integración
- ✓ Tests con supertest
- ✓ Testing de elementos avanzados

Objetivos de la clase

- Conocer sobre tests unitarios
- Realizar test unitarios con assert y Mocha
- Realizar test con cadenas de lenguaje con Mocha + Chai

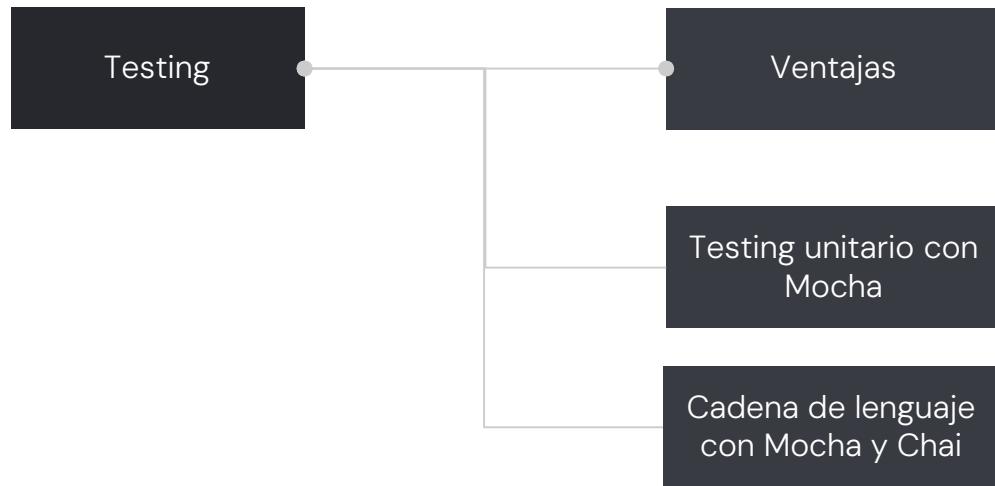
CLASE N°8

Glosario

Swagger: es una herramienta de documentación de código, la cual nos permitirá mantener cada módulo de nuestra API dentro de un **espectro de entendimiento sólido.**



MAPA DE CONCEPTOS



Testing

CODERHOUSE

Mi aplicación ya funciona, ¿y ahora qué?

Cuando desarrollamos, lo primero que hacemos es pensar en el resultado del producto final, lo cual nos lleva a realizar un desarrollo, en ocasiones, más acelerado de lo que se debería e ignorando ciertas partes de dicho proceso.

Una de las partes más ignoradas por los desarrolladores es la parte de **las pruebas**. Y con ignorar no nos referimos a no hacer ninguna prueba, sino que la forma de hacerlas es bastante descontrolada y desinteresada, ya que no se plantean pruebas correctamente estructuradas para reducir en gran medida la posibilidad de error.

Ventajas de realizar testing

Realizar testing puede llevarnos a múltiples beneficios, tales como:

- ✓ Reducción de posibilidad de error: Es el objetivo principal del testing, el considerar posibilidades para poder subsanarlas antes de que lleguen al cliente.
- ✓ Incremento en el conocimiento del código desarrollado: Pensar con detenimiento y hacer con detalle un flujo de pruebas nos puede llevar a comprender mejor el contexto aplicado del módulo y no sólo su funcionalidad, permitiendo mejoras a futuro.
- ✓ Descubrimiento de puntos ciegos del código: ¿Alguna vez te has enfrentado a un "Lo hubiera sabido antes"? Repasar el flujo testeado permitirá llegar a estos casos lo antes posible, permitiendo que podamos atenderlo con antelación.
- ✓ Posibilidad de refactoring: Cuando hacemos pruebas, repasamos constantemente un flujo, lo cual también nos permite notar aspectos que podríamos mejorar el flujo.

Dos formas de pensar en test

El testing es un tema de cuidado, y por lo tanto hay que **tomarlo en consideración a partir de dos criterios.**

- ✓ Testing unitario
- ✓ Testing de integración

Un test de integración es evidentemente más complicado, el día de hoy nos centraremos a los tests unitarios.

CODERHOUSE

Test unitario

Un test unitario está pensado para funcionalidades aisladas, es decir, aquellas funcionalidades en las que **no se consideran el contexto u otros componentes**.

La unidad es el elemento más pequeño que hay, de manera que construir unidades y testearlas será bastante sencillo.

Test unitario

Por ejemplo, podemos hacer un testing unitario sobre el **dao de una entidad en particular**.

Podemos probar cosas como:

- ✓ Que el módulo lea correctamente los datos de la entidad en el sistema de persistencia.
- ✓ Que el módulo escriba correctamente los datos de la entidad en el sistema de persistencia.
- ✓ Que el módulo actualice datos correctamente
- ✓ Que el módulo elimine datos correctamente.

Operaciones sencillas para un módulo sencillo.

Al final, no refleja la funcionalidad completa de una entidad (como el conjunto de su router, servicio y dao)

Mocha

CODERHOUSE

Mocha

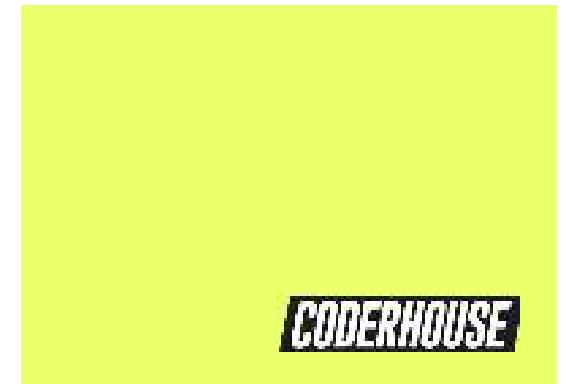
Es un framework de testing originalmente diseñado para nodejs, el cual nos permitirá ejecutar entornos completos para poder hacer cualquier tipo de pruebas que necesitemos.

Para poder comenzar a utilizarlo, primero hay que tenerlo instalado en nuestro entorno.

¿Qué entorno utilizaremos? Para esto, reutilizaremos el proyecto de la clase previa **Adoptme**. ¡Tenlo a la mano para poder trabajar en la clase de hoy!



simple, flexible, fun



Terminología elemental de testing

Assert: módulo nativo de nodejs que nos permitirá hacer validaciones de manera estricta.

archivo.test.js: la subextensión .test.js indica que el archivo será utilizado dentro de un contexto de testing

describe: función utilizada para definir diferentes contextos de testeo, podemos tener la cantidad de contextos que deseemos en un flujo de testing, siempre y cuando reflejen intenciones diferentes.

it: unidad mínima de nuestro testing, en ella, definimos qué acción se está realizando y cuál será el resultado esperado.

Terminología elemental de testing

before: Función que nos permite inicializar elementos antes de comenzar con todo el contexto de testeo.

beforeEach: Función que nos permite inicializar elementos antes de comenzar **cada test** dentro de un contexto particular.

after: Función que nos permite realizar alguna acción una vez finalizado el contexto de testeo

afterEach: Función que nos permite realizar alguna acción una vez finalizado **cada test** dentro del contexto particular.



Checkpoint: Espacio de preparación

Toma un tiempo para clonar el proyecto [aquí](#). Revisa el código e instala las dependencias indicadas.

Tiempo estimado: **3 minutos**

CODERHOUSE

Añadimos a nuestro proyecto la capacidad de testear

Mocha es una dependencia externa, así que procederemos a instalarla con el comando:

```
npm install -D mocha
```

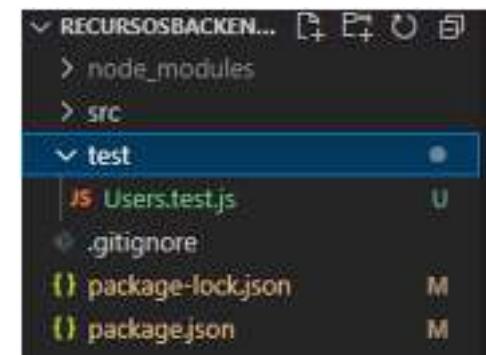
La razón de instalarlo dentro de nuestras dependencias de desarrollo es que el testing sólo se realiza antes de entrar en un entorno productivo. Cuando nuestro proyecto se encuentre desplegado en la nube, no habría necesidad de querer correr un test en éste.

CODERHOUSE

Comenzamos a estructurar nuestro primer módulo de testing

En diapositivas pasadas mencionamos que podíamos ver como primer ejemplo el uso de un Dao, así que vamos a utilizar alguno de los que nos provee el proyecto. Para este caso, tomaremos el Dao de User.

Para poder crear un módulo de testing, vamos a crear una carpeta fuera de la carpeta src llamada **test**. Dentro de ésta tendremos un archivo por cada módulo que deseamos testear



¡Todo listo para comenzar a escribir nuestros tests!

Primer test

CODERHOUSE

Obtengamos todos los elementos necesarios para nuestras pruebas

Lo primero en nuestro archivo es importar los módulos necesarios para el test a realizar.

Como en este caso queremos probar un Dao, necesitaremos mongoose para poder inicializar la conexión a nuestra base, seguido del Dao a utilizar.

Ahora inicializamos la conexión de mongoose para este flujo de pruebas y configuramos Assert de nodejs para poder evaluar los tests en strict mode

```
JS Users.test.js U ×  
test > JS Users.test.js > [0] assert  
1 import mongoose from 'mongoose';  
2 import User from '../src/dao/Users.da  
3 import Assert from 'assert';  
4  
5 mongoose.connect('mongodb+srv://Coder:  
6  
7 const assert = Assert.strict;  
8
```

Describe: entramos en un contexto

```
describe('Testing Users Dao', () => {
  before(function () {
    this.usersDao = new User()
  })
  beforeEach(function(){
    this.timeout(5000);
  })
})
```

Describe es la primera parte informativa de nuestro test, el cual indica de manera explícita **cuál será el módulo a testear**. Todo lo que coloquemos dentro de este describe pertenece al mismo contexto de test.

Ahora, utilizaremos una función **before** para poder contar con una variable *usersDao*, la cual nos servirá para utilizar en todos nuestros tests futuros.

Además, gracias al **beforeEach**, podremos colocar un tiempo máximo de resolución (por defecto son 2 segundos). Ya que estamos utilizando una base de datos, se recomienda colocar un tiempo de reesolución máximo más elevado.

¿Qué debe hacer nuestro primer test?

Con la palabra **it**, seremos capaces de describir qué es lo que se espera de la operación que se realizará en dicha prueba.

Por ejemplo, si queremos probar el método get del Dao, podríamos escribir:

it ("El get debe devolver un arreglo")

Así, tenemos una idea de lo que realiza o no realiza el programa.

```
 4 Example 11 X
 5
 6 /**
 7 * Descripción: Se despliega una página web y se verifica si el título coincide con el que se ha establecido en la URL.
 8 */
 9
10 import org.openqa.selenium.WebDriver;
11 import org.openqa.selenium.chrome.ChromeDriver;
12 import org.openqa.selenium.WebElement;
13
14 WebDriver navegador = null;
15
16 navegador.get("http://www.santander.com");
17
18 WebElement titulo = navegador.getTitle();
19
20 System.out.println(titulo);
21
22 assertEquals("SANTANDER | Santander", titulo);
23
24
25 navegador.quit();
26
27
28 // Verifica que el título de la página coincide con el que se ha establecido en la URL.
```

Uso de assert

con **assert**, podremos hacer las operaciones que determinarán si un test pasa o no. En este caso, comparamos si el tipo devuelto por el método get() del Dao efectivamente es un array

```
it('El Dao debe poder obtener los usuarios en formato de arreglo',async function(){
    console.log(this.usersDao);
    //Nota cómo el callback representa el entorno a ejecutar.
    //Este entorno es aislado, por lo que no afectará a las demás pruebas.
    const result = await this.usersDao.get();
    assert.strictEqual(Array.isArray(result),true);
})
```

Resultado de un test

Una prueba no tiene términos medios, al final:

Si una prueba pasa, podremos visualizarlo de la siguiente forma:

```
> plantilladocumentacion@1.0.0 test
> mocha test/Users/test.js

Testing Users Dao
  ✓ El Dao debe poder obtener los usuarios en formato de arreglo (1473ms)

  1 passing (1s)
```

Resultado de un test

Por otra parte cuando un test no pase, nos indicará cuál fue el resultado que esperábamos (en color verde), contra el resultado que realmente recibimos por parte de nuestro código (en color rojo) (para este test, se cambió el código de assert por **false**)

```
Testing Users Dao
  1) El Dao debe poder obtener los usuarios en formato de arreglo

  0: passing (is)
  1: failing

  1) Testing Users Dao
    El Dao debe poder obtener los usuarios en formato de arreglo:
      AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
      true != false
        + expected - actual
        -true
        +false
```



Ejemplo en vivo

Se desarrollará un conjunto de tests para el resto del Dao de Usuarios. Se abordarán diferentes formas de utilizar el assert

Se debe cumplir el siguiente listado de tests:



CODERHOUSE



Ejemplo en vivo

- ✓ El Dao debe agregar correctamente un elemento a la base de datos.
- ✓ Al agregar un nuevo usuario, éste debe crearse con un arreglo de mascotas vacío por defecto.
- ✓ El Dao puede obtener a un usuario por email

Duración: **15 minutos**

¡Importante!

Si estás teniendo problemas debido a que el usuario se queda en la base de datos a lo largo de tu test, y quieres borrarlo cada vez que inicie el entorno o cada test, puedes agregar al `beforeEach` la directiva para que vacíe la colección de usuarios en cada momento que inicie un test:

```
beforeEach(function () {
  mongoose.connection.collections.users.drop();
  this.timeout(5000);
})
```

Ejemplo: Test 1

```
it('El Dao debe agregar un usuario correctamente a la base de datos', async function () {
  let mockUser = {
    first_name: 'Coder',
    last_name: 'House',
    email: 'correo@correo.com',
    password: "123",
  }
  const result = await this.usersDao.save(mockUser);
  //assert.ok evaluará si el parámetro pasado es truthy, es decir, que sea cualquier valor definido
  //Que no se pueda tomar por falso. (En este caso, el test pasará si _id está definido)
  assert.ok(result._id);
})
```

Ejemplo: Test 2

```
it('El Dao agregará al documento insertado un arreglo de mascotas vacío por defecto', async function () {
  let mockUser = {
    first_name: 'Coder',
    last_name: 'House',
    email: 'correo@correo.com',
    password: "123",
  }
  const result = await this.usersDao.save(mockUser);
  //assert.deepEqual hace referencia a una comparación interna y profunda (es decir, incluyendo)
  //sus propiedades internas. Si se evita la palabra "deep", el test enviará error debido a que estará
  //valorando que sean referencias distintas.
  assert.deepEqual(result.pets, [])
})
```

Ejemplo: Test 3

```
it('El Dao puede obtener a un usuario por email', async function () {
  let mockUser = {
    first_name: 'Coder',
    last_name: 'House',
    email: 'correo@correo.com',
    password: "123",
  }
  const result = await this.usersDao.save(mockUser);

  const user = await this.usersDao.getBy({ email: result.email });
  assert.strictEqual(typeof user, 'object');
})
```



Break

¡10 minutos y volvemos!

CODERHOUSE

Complemento de assertions: Chai

CODERHOUSE

Chai

Chai es una librería de **assertions**, la cual nos permitirá realizar comparaciones de test más *claras*.

Está pensado para que, las evaluaciones de test que se hagan en cada módulo, sean lo más legibles posibles, haciendo que sean lo más apoyadas al inglés, reduciendo el nivel de abstracción.

Chai trabaja en un modelo de assertion extendido también, sin embargo, en esta clase nos centraremos en aplicar el enfoque de comportamiento (BDD) a partir de su módulo de **cadena de lenguaje**



Chai Assertion Library

CODERHOUSE

Cadenas de lenguaje de Chai

Chai permitirá conectar palabras del inglés, con el fin de poder realizar una prueba más entendible, algunos de estos conectores son:

- ✓ **to**: conector inicial para armar la frase.
- ✓ **be**: para identificar que el elemento **sea** algo en particular.
- ✓ **have**: para corroborar que el valor a evaluar **tenga** algo.
- ✓ **and**: para encadenar validaciones.

Cadenas de lenguaje de Chai

Estas cadenas de lenguaje se conectan con operadores más específicos, como:

- ✓ **not**: para realizar una negación.
- ✓ **deep**: Para evaluaciones profundas.
- ✓ **equal**: para hacer una comparación de igualdad.
- ✓ **property**: para apuntar a alguna propiedad de un objeto.

Puedes ver la lista de cadenas de lenguaje y operadores en [este link](#)

Comenzar a trabajar utilizando Chai

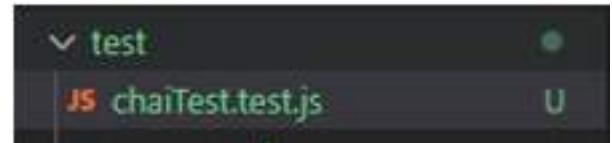
Primero instalaremos chai en el mismo proyecto en el que estamos trabajando.

Entiendo que chai es un complemento para testing, no será necesario hacer la instalación completa, bastará hacerlo para desarrollo

```
npm install -D chai
```

```
npm install -D chai
```

Después, crearemos otro archivo llamado chaiTest.test.js, el cual replicará las pruebas del otro archivo, pero esta vez las traduciremos a chai.



CODERHOUSE

Imports

Volveremos a importar mongoose para la conexión y Users para el Dao, sin embargo, esta vez vamos a importar chai para poder comenzar a utilizarlo.

Hay tres modelos para trabajar con chai: **chai expect**, **chai should** y **chai assert**, en este caso particular utilizaremos chai.expect

```
js chaiTest/test.js: U X
test 1: chaiTest/test.js:3...
  1 import chai from 'chai';
  2 import mongoose from 'mongoose';
  3 import Users from './src/dao/Users.dao.js';
  4
  5 //Utilizaremos la variable expect a partir de ahora para hacer nuestras comparaciones.
  6 const expect = chai.expect;
  7
  8 mongoose.connect('URL DE MONGO')
```

Colocamos el cuerpo inicial

Recordando la arquitectura:

```
describe('Set de tests con Chai', ()=>{
  before(function () {
    this.usersDao = new Users()
  })
  beforeEach(function () {
    mongoose.connection.collections.users.drop();
    this.timeout(5000);
  })
})
```

describe define el significado del test.

before sirve para ejecutarse antes de iniciar todo el flujo de testing, en este caso lo utilizamos para inicializar el Dao de usuarios. **beforeEach** se ejecuta antes de cada test, de manera que lo utilizamos para limpiar la colección y para setear un tiempo máximo de resolución (porque estamos trabajando con bases de datos)

Probando la sintaxis de chai

Analicemos la forma en la que se construyó el primer test

```
it('El Dao debe poder obtener los usuarios en formato de arreglo',async function(){
  const result = await this.usersDao.get();
  expect(result).to.be.deep.equal([]);
})
```

expect() recibe como parámetro el valor que estamos por testear, posterior a esto, hacemos conexiones de palabras con el fin de llegar a la pregunta final (preguntar si es un arreglo). Nota cómo se “**naturaliza**” el lenguaje en el cual preguntamos las cosas, de manera que, para lecturas de pruebas, resulta bastante sencillo.

Al ser un encadenamiento de lenguaje, se puede llegar al mismo resultado con otras variantes como:

```
expect(result).deep.equal([]);
expect(Array.isArray(result)).to.be.ok;
expect(Array.isArray(result)).to.be.equals(true);
```



Replanteamiento de tests pasados + update/delete

Duración: 20 min



ACTIVIDAD EN CLASE

Replanteamiento de tests pasados

Descripción de la actividad.

Con base en el ejemplo desarrollado en vivo previamente, replantear los tests realizados con **assert**, recuerda que puedes apoyarte de la documentación [aquí](#)

Además, realizar un test que evalúe que el método update y delete del Dao de usuarios sea efectivo, puedes utilizar el método de validación que deseas.



Hands on lab

En esta instancia de la clase **repasaremos** algunos de los conceptos vistos en clase con una aplicación

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **20 minutos**

CODERHOUSE



HANDS ON LAB

Test de más elementos aislados del proyecto

Con base en el proyecto que tenemos de Adoptme, se nos solicita realizar un proceso de testing para las utilidades de bcrypt y la funcionalidad del DTO. Los elementos que nos solicitan validar son:

- ✓ El servicio debe realizar un hasheo efectivo de la contraseña (debe corroborarse que el resultado sea diferente a la contraseña original)
- ✓ El hasheo realizado debe poder compararse de manera efectiva con la contraseña original (la comparación debe resultar en true)
- ✓ Si la contraseña hasheada se altera, debe fallar en la comparación de la contraseña original.
- ✓ Por parte del DTO de usuario: Corroborar que el DTO unifique el nombre y apellido en una única propiedad. (Recuerda que puedes evaluar múltiples expects)
- ✓ Por parte del DTO de usuario: El DTO debe eliminar las propiedades innecesarias como password, first_name, last_name.

¿Preguntas?

CODERHOUSE

Muchas gracias.

CODERHOUSE

Resumen de la clase hoy

- ✓ Testing
- ✓ Test unitario con Mocha
- ✓ Cadena de lenguaje con Mocha + Chai

**Opina y valora
esta clase**

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Guía de Actividades hacia el Proyecto Final

Testing y Escalabilidad Backend

CODERHOUSE

¡Bienvenidas y bienvenidos!

Qué bueno encontrarlos/as en este espacio, el cual hemos creado para que puedan conseguir en un mismo lugar, de manera rápida y ágil, todos los desafíos entregables que plantea el módulo II.

A continuación presentamos el sistema de entregas de los cursos de Coder. Luego, en un tablero, podrán ver **las 14 clases establecidas en este curso**, marcando con el ícono correspondiente las clases que sí tienen entregables.

De esta forma podrás tener un pantallazo del cronograma de clases y los desafíos que deberás completar.

Instancias prácticas

Conoce el sistema de entregas de Coder



Actividades de clase

- ✓ Tareas que ponen en práctica lo trabajado en cada encuentro.
- ✓ Se resuelven en clase.



Prácticas hacia el Proyecto Final

- ✓ Actividades relacionadas al PF.
- ✓ Se resuelven fuera de clase.
- ✓ Se encuentran en esta guía de actividades.
- ✓ Te ayudarán a construir las pre-entregas progresivamente.
- ✓ Te recomendamos hacerlas todas ya que nutren directamente a tu PF.

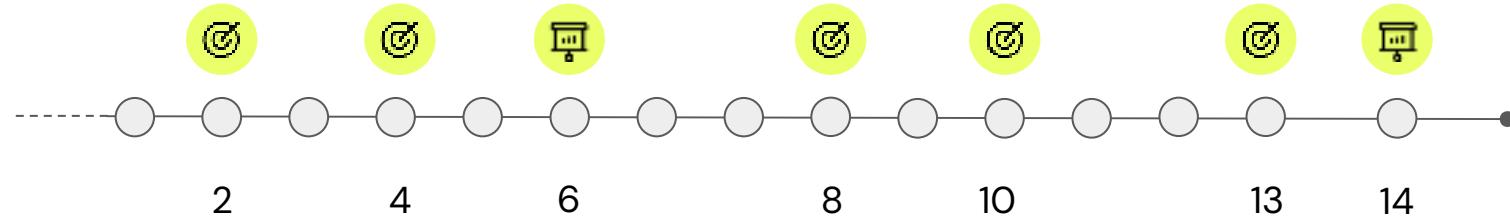


Preentregas

- ✓ Entregables con el estado de avance del **Proyecto Final** para ser corregidas por tu tutor/a.
- ✓ Son obligatorias.
- ✓ Deben cargarse a la plataforma.



GRILLA DE ENTREGAS



CLASE 2

Optimización

CODERHOUSE



Mocking y manejo de errores

CODERHOUSE



ACTIVIDAD PRÁCTICA

Mocking y manejo de errores

Consigna

- ✓ Se aplicará un módulo de mocking y un manejador de errores al proyecto [Adoptme](#).

Formato

- ✓ Link al repositorio de github sin node_modules

Sugerencias

- ✓ Céntrate solo en los errores más comunes

Aspectos a incluir

- ✓ Crear un módulo de Mocking para el servidor, con el fin de generar mascotas (sin **owner** y con **adopted** en “false”) de acuerdo a un parámetro numérico. Utilizar este módulo en un endpoint GET llamado “/mockingpets” y generar 100 mascotas con el mismo formato que entregaría una petición de Mongo.
- ✓ Además, generar un customizador de errores y crear un diccionario para tus errores más comunes al registrar un usuario, crear una mascota, etc.

CLASE 4

Logging y performance

CODERHOUSE



Implementación de logger

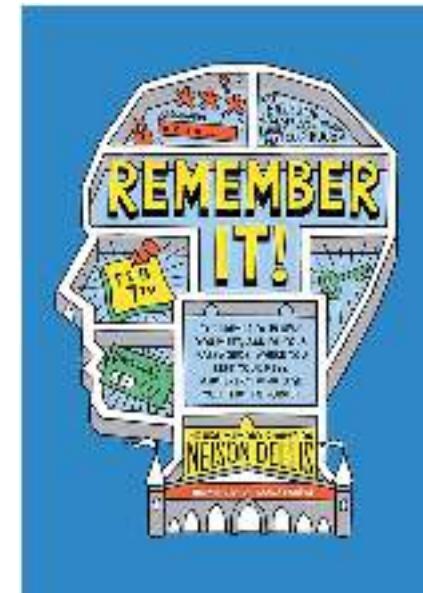
CODERHOUSE

Recordemos...

**Basado en el entregable
de la clase 2**

Revisamos y analizamos el
proyecto Adoptme.

Realizamos un servicio de mocking
en dicho proyecto.



Ahora... Agregaremos un logger para tener mejores registros



ACTIVIDAD PRÁCTICA

Implementación de logger

Consigna

- ✓ Basado en nuestro [proyecto principal](#), implementar un logger.

Aspectos a incluir

- ✓ Primero, definir un sistema de niveles que tenga la siguiente prioridad (de menor a mayor):
`debug, http, info, warning, error, fatal`
- ✓ Después implementar un logger para desarrollo y un logger para producción, el logger de desarrollo deberá loggear a partir del nivel debug, sólo en consola

- ✓ Sin embargo, el logger del entorno productivo debería loggear sólo a partir de nivel info.
- ✓ Además, el logger deberá enviar en un transporte de archivos a partir del nivel de error en un nombre “errors.log”
- ✓ Agregar logs de valor alto en los puntos importantes de tu servidor (errores, advertencias, etc) y modificar los `console.log()` habituales que tenemos para que muestren todo a partir de winston.
- ✓ Crear un endpoint “/loggerTest” que permita probar todos los logs

CODERHOUSE



ACTIVIDAD PRÁCTICA

Implementación de logger

Formato

- ✓ link al repositorio de Github con el proyecto sin node_modules

Sugerencias

- ✓ Puedes revisar el testing del entregable [Aquí](#)
- ✓ La ruta “/loggerTest” es muy importante para que tu entrega pueda ser calificada de manera rápida y eficiente. ¡No olvides colocarla!

CLASE 6

Pre entrega de tu proyecto final

CODERHOUSE



**Pre entrega de tu
Proyecto Final**

CODERHOUSE



ENTREGA DEL PROYECTO FINAL

Primera entrega

Se debe entregar

- ✓ Crear un router llamado mocks.router.js que funcione bajo la ruta base **/api/mocks**.
 - ✓ Mover el endpoint “/mockingpets” (Desarrollado en el primer Desafío Entregable) dentro de este router.
 - ✓ Crear un módulo de Mocking para generar usuarios de acuerdo a un parámetro numérico. Dichos usuarios generados deberán tener las siguientes características:
 - En “password” debe tener la contraseña “coder123” encriptada.
 - “role” puede variar entre “user” y “admin”.
 - “pets” debe ir como array vacío.
- ✓ Dentro del router mocks.router.js, utilizar este módulo en un endpoint GET llamado “/mockingusers”, y generar 50 usuarios con el mismo formato que entregaría una petición de Mongo.



ENTREGA DEL PROYECTO FINAL

Primera entrega

Se debe entregar

- ✓ Dentro del router mocks.router.js, desarrollar un endpoint POST llamado **/generateData** que reciba los parámetros numéricos “users” y “pets” para generar e insertar en la base de datos la cantidad de registros indicados.
- ✓ Comprobar dichos registros insertados mediante los servicios GET de users y pets

Formato

- ✓ Link al repositorio de Github con el proyecto completo, sin la carpeta de Node_modules.

CLASE 8

Documentación de API

CODERHOUSE



Documentar API

CODERHOUSE



ACTIVIDAD PRÁCTICA

Documentar API

Consigna

- ✓ Realizar la configuración necesaria para tener documentado tu [proyecto principal](#) a partir de Swagger.

Aspectos a incluir

- ✓ Se debe tener documentado los módulos de:
 - Sessions.
 - Pets.
 - Adoptions.

Formato

- ✓ Link al repositorio de Github sin node_modules

Sugerencias

- ✓ Recuerda que es un proceso de documentación, ¡Hay que ser lo más claros posibles!

CLASE 10

Testing avanzado

CODERHOUSE



Módulos de Testing para el proyecto “Adoptme”

CODERHOUSE



ACTIVIDAD PRÁCTICA

Módulos de testing para proyecto final

Consigna

- ✓ Realizar módulos de testing para tu [proyecto principal](#), utilizando los módulos de mocha + chai + supertest.

Aspectos a incluir

- ✓ Se deben desarrollar los tests para todos los endpoints de:
 - Router de users.
 - Router de pets.
- ✓ NO desarrollar únicamente tests de status, la idea es trabajar lo mejor desarrollado posible las validaciones de testing

Formato

- ✓ link del repositorio en github sin node_modules

Sugerencias

- ✓ Ya que el testing lo desarrollarás tú, no hay una guía de test por leer. ¡Aplica tu mayor creatividad en tus pruebas!

CLASE 13

Práctica integradora

CODERHOUSE



Práctica de integración sobre tu proyecto principal

CODERHOUSE



ACTIVIDAD PRÁCTICA INTEGRADORA

Consigna

Con base en el [proyecto principal](#) que venimos desarrollando, toca solidificar algunos procesos.

Aspectos a incluir

- ✓ Modificar el modelo de User para que cuente con una nueva propiedad “documents” el cual será un array que contenga los objetos con las siguientes propiedades
 - name: String (Nombre del documento).
 - reference: String (link al documento).

No es necesario crear un nuevo modelo de Mongoose para éste.
- ✓ Además, agregar una propiedad al usuario llamada “last_connection”, la cual deberá modificarse cada vez que el usuario realice un proceso de login y logout



ACTIVIDAD PRÁCTICA INTEGRADORA

Aspectos a incluir

- ✓ Crear un endpoint en el router de usuarios **api/users/:uid/documents** con el método POST que permita subir uno o múltiples archivos y actualizar el atributo “documents” del usuario en cuestión. Utilizar el middleware de Multer para poder recibir los documentos que se carguen en el proyecto.
- ✓ El middleware de multer deberá estar modificado para que pueda guardar en diferentes carpetas los diferentes archivos que se suban.
 - Si se sube una imagen de una mascota, deberá guardarlo en una carpeta **pets**, mientras que ahora al cargar un documento, multer los guardará en una carpeta **documents**.
- ✓ Desarrollar los tests funcionales para los endpoints de **api/sessions/register** y **api/sessions/login** utilizando los módulos de mocha, chai y supertest.



ACTIVIDAD PRÁCTICA INTEGRADORA

Formato

- ✓ Link al repositorio de GitHub con el proyecto completo (no incluir node_modules).

Sugerencias

- ✓ Para el uso de Multer, se puede tomar de referencia el endpoint **api/pets/withimage** que ya utiliza Multer para subir las imágenes de las mascotas.

CLASE 14

Product Cloud: Despliegue de nuestro aplicativo

CODERHOUSE



Dockerizando nuestro Proyecto

CODERHOUSE



ENTREGA DEL PROYECTO FINAL

Dockerizando nuestro Proyecto

Objetivos generales

- ✓ Implementar las últimas mejoras en nuestro proyecto y Dockerizarlo.

Objetivos específicos

- ✓ Documentar las rutas restantes de nuestro proyecto.
- ✓ Añadir los últimos tests
- ✓ Crear una imagen de Docker.

Se debe entregar

- ✓ Documentar con Swagger el módulo de "Users".

Se debe entregar

- ✓ Desarrollar los tests funcionales para todos los endpoints del router "adoption.router.js".
- ✓ Desarrollar el Dockerfile para generar una imagen del proyecto.
- ✓ Subir la imagen de Docker a Dockerhub y añadir en un ReadMe.md al proyecto que contenga el link de dicha imagen.



ENTREGA DEL PROYECTO FINAL

Dockerizando nuestro Proyecto

Formato

- ✓ Link al repositorio de Github con el proyecto (sin node_modules)
- ✓ Además, archivo .env para poder correr el proyecto.

Sugerencias

- ✓ Para repasar Docker, se recomienda revisar la clase 5 “Clusters & Escalabilidad”.

¡Éxitos!

CODERHOUSE

Educación digital
para el mundo **real.**

CODERHOUSE

Material complementario

CODERHOUSE

Cuarta práctica integradora

Guía de Skills

Skills para Logging

- ✓ Comprender la importancia de utilizar un logger
- ✓ Comprender el uso de Winston Logger y aplicarlo
- ✓ Entender los diferentes tipos de transportes
- ✓ Entender sobre los niveles de logging
- ✓ Configurar nuestros propios niveles de logging

Skills para documentación

- ✓ Comprender la importancia de documentar
- ✓ Comprender el uso de Swagger para documentación
- ✓ Realizar la Swaggerización por archivos de cada Módulo
- ✓ Comprender sobre los elementos que compone un módulo Swaggerizado (schemas, inputs, requestBodies, responses, etc)

Skills para testing

- ✓ Comprender sobre módulos de Testing
- ✓ Conocimientos de realización de Testing unitario
- ✓ Conocimiento sobre Test de integración
- ✓ Uso de Mocha
- ✓ Uso de Chai
- ✓ Uso de SuperTest

Material complementario

CODERHOUSE

Guía de skills

3era práctica integradora

Skills para Arquitectura por capas

- ✓ Comprender el manejo de arquitectura por capas
- ✓ Entender e isolar la responsabilidad de ruteo
- ✓ Entender e isolar la responsabilidad de controlador
- ✓ Entender e isolar la responsabilidad de servicio
- ✓ Entender e isolar la responsabilidad de persistencia

Skills para patrones de diseño

- ✓ Entender e implementar el patrón de diseño Data Access Object (DAO)
- ✓ Entender e implementar el patrón de diseño Data Transfer Object (DTO)
- ✓ Entender e implementar el patrón de diseño Repository
- ✓ Entender e implementar el patrón de diseño Factory

Skills para Mailing

- ✓ Comprender sobre el protocolo SMTP
- ✓ Estructurar correos básicos a partir de una cuenta configurada de gmail.
- ✓ Implementar attachments en un correo electrónico.

Skills para entornos

- ✓ Entender la diferencia entre un entorno de desarrollo y un entorno de producción.
- ✓ Configurar entornos a partir de .dotenv
- ✓ Setear variables de entorno y un archivo general de configuración del aplicativo.

Material complementario

CODERHOUSE

Cuarta práctica integradora

Guía de Skills

Skills para Logging

- ✓ Comprender la importancia de utilizar un logger
- ✓ Comprender el uso de Winston Logger y aplicarlo
- ✓ Entender los diferentes tipos de transportes
- ✓ Entender sobre los niveles de logging
- ✓ Configurar nuestros propios niveles de logging

Skills para documentación

- ✓ Comprender la importancia de documentar
- ✓ Comprender el uso de Swagger para documentación
- ✓ Realizar la Swaggerización por archivos de cada Módulo
- ✓ Comprender sobre los elementos que compone un módulo Swaggerizado (schemas, inputs, requestBodies, responses, etc)

Skills para testing

- ✓ Comprender sobre módulos de Testing
- ✓ Conocimientos de realización de Testing unitario
- ✓ Conocimiento sobre Test de integración
- ✓ Uso de Mocha
- ✓ Uso de Chai
- ✓ Uso de SuperTest