

# Esta clase va a ser

- grabada

Certificados oficialmente por

 PedidosYa

**CODERHOUSE**

Clase 40. PROGRAMACIÓN BACKEND

# Testing unitario

Certificados oficialmente por



**CODERHOUSE**

# Temario

39

## Documentación

- ✓ Importancia de la documentación
- ✓ Documentar con Swagger

40

## Testing Unitario

- ✓ [Módulos de testing](#)
- ✓ [Testing con Mocha](#)
- ✓ [Testing con Chai](#)

41

## Testing Avanzado

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

# Objetivos de la clase

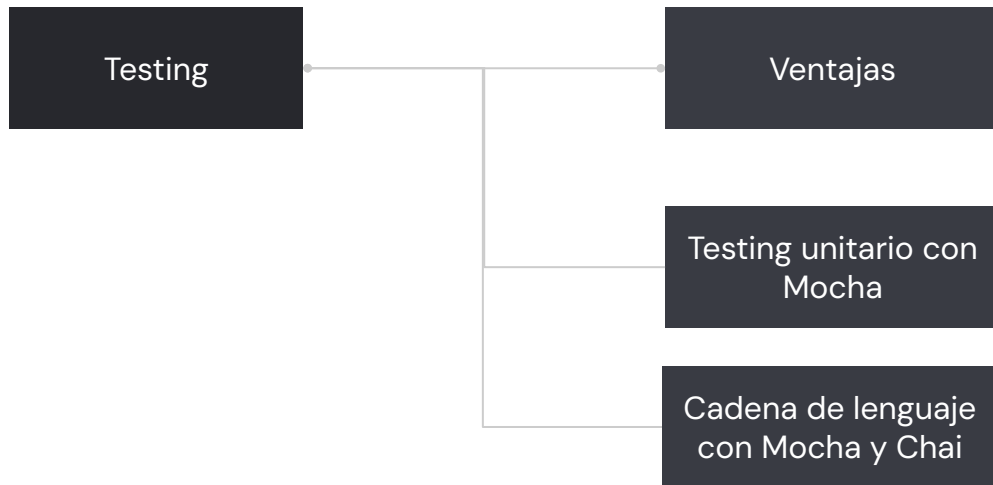
- Conocer sobre tests unitarios
- Realizar test unitarios con assert y Mocha
- Realizar test con cadenas de lenguaje con Mocha + Chai

CLASE N°39

# 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

# 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.

# 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

# 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

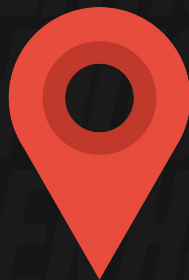
**before:** Función que nos permite inicializar elementos antes de comenzar con todo el contexto de testeo.

**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**

# 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
```

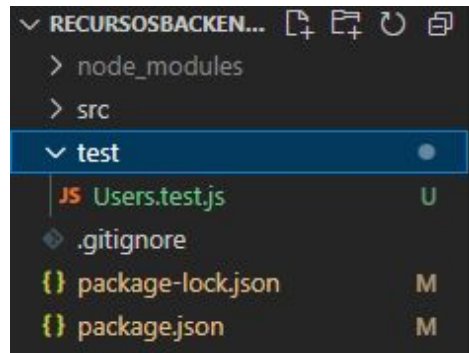
La razón de instalarlo dentro de nuestras dependencias de desarrollo es que el testing sólo se realiza antes de entrar en un entorno productivo. Cuando nuestro proyecto se encuentre desplegado en la nube, no habría necesidad de querer correr un test en éste.

# Comenzamos a estructurar nuestro primer módulo de testing

En las diapositivas anteriores 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!



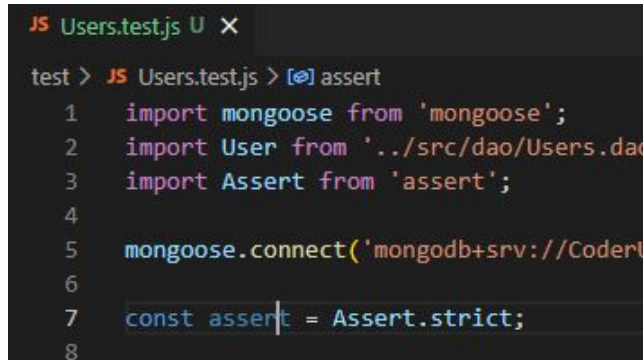
# Primer test

# 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

A screenshot of a code editor window titled 'JS Users.test.js U X'. The editor shows a terminal-like interface with the command 'test > JS Users.test.js > [icon] assert'. Below the command, the code is as follows:

```
1 import mongoose from 'mongoose';
2 import User from '../src/dao/Users.dao';
3 import Assert from 'assert';
4
5 mongoose.connect('mongodb+srv://CoderHouse:password@cluster0.mongodb.net/test?retryWrites=true&w=majority');
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.

```
JS Users.test.js U X
test > JS Users.test.js > describe('Testing Users Dao') callback > it('El Dao debe poder obtener los usuarios en
1 import mongoose from 'mongoose';
2 import User from '../src/dao/Users.dao.js';
3 import assert from 'assert';
4
5 mongoose.connect('AQUÍ TU URL APUNTANDO A UNA BASE DE TEST')
6
7
8 describe('Testing Users Dao', () => {
9   before(function () {
10     this.usersDao = new User()
11   })
12   it('El Dao debe poder obtener los usuarios en formato de arreglo', () => {
13     //Nota cómo el callback representa el entorno a ejecutar.
14     //Este entorno es aislado, por lo que no afectará a las demás pruebas.
15   })
16 })
```

# 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 (1s)
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:





## 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: 'correoprueba@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: 'correoprueba@correo.com',  
    password: "123",  
  }  
  const result = await this.usersDao.save(mockUser);  
  //assert.deepStrictEqual 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.deepStrictEqual(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: 'correoprueba@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!

# Complemento de assertions: Chai

# 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 apegadas 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**



# 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

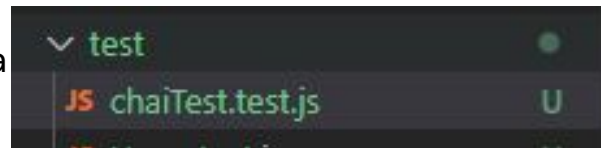
Primero instalaremos chai en el mismo proyecto en el que estamos trabajando.

Entendiendo 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.



# 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 > JS chaiTest.test.js > ...
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')
9
```

# Colocamos el cuerpo inicial

```
describe('Set de tests con Chai', () => {  
  before(function () {  
    this.usersDao = new Users()  
  })  
  beforeEach(function () {  
    mongoose.connection.collections.users.drop();  
    this.timeout(5000);  
  })  
})
```

Recordando la arquitectura:

**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 desees.



## 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**

# 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 hash efectivo de la contraseña (debe corroborarse que el resultado sea diferente a la contraseña original)
- ✓ El hash realizado debe poder compararse de manera efectiva con la contraseña original (la comparación debe resultar en true)
- ✓ Si la contraseña hashada 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?

**#DemocratizandoLaEducación**

**Muchas gracias.**



# Resumen

## de la clase hoy

- ✓ Testing
- ✓ Test unitario con Mocha
- ✓ Cadena de lenguaje con Mocha + Chai

**Opina y valora**  
**esta clase**