

# Esta clase va a ser

- grabada

Certificados oficialmente por

 PedidosYa

**CODERHOUSE**

Clase 40. PROGRAMACIÓN BACKEND

# Testing avanzado

Certificados oficialmente por



**CODERHOUSE**

# Temario

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](#)

42

## 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°40

# 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**.

CLASE N°40

# 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.

## MAPA DE CONCEPTOS



# Test unitario vs Test de integración



# Sobre el test unitario

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

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**.



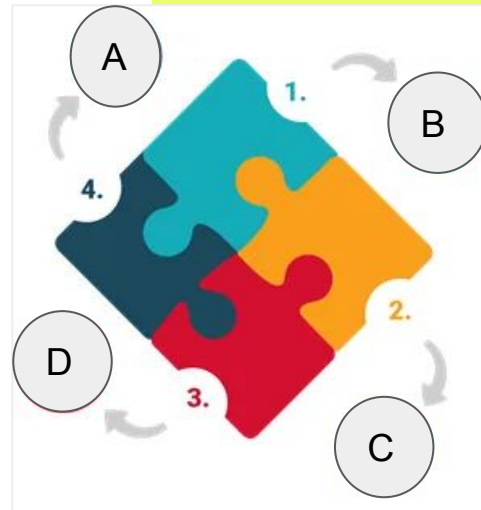
# Sobre el testing de integración

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**.

# Testings de integración y funcionales con Supertest

# 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.



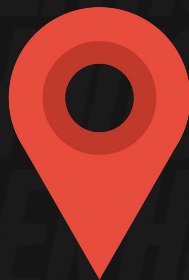
# Inicializando un test con supertest

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

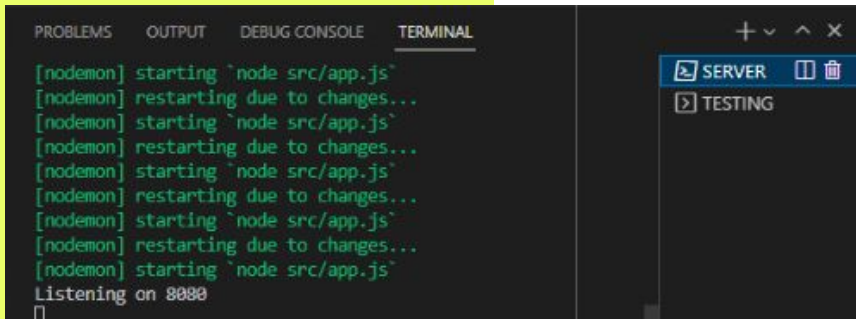
Toma un tiempo para clonar o inicializar el proyecto **Aquí**. Revisa el código e instala las dependencias indicadas.

Tiempo estimado: **3 minutos**



# Preparamos una doble terminal

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



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[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`
Listening on 8080
```

SERVER TESTING

- ✓ 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 supertest.test.js U X
test > JS supertest.test.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 describe: 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)

JS supertest.test.js U X

test > JS supertest.test.js > describe('Testing Adoptme') callback > describe('Test de mascotas') callback > it('El endpoint POST /api/pets debe crear una

```
7 describe('Testing Adoptme', ()=>{
8   describe('Test de mascotas', ()=>{
9     it('El endpoint POST /api/pets debe crear una mascota correctamente', async()=>{
10       const petMock = {
11         name: "Patitas",
12         especie: "Pez",
13         birthDate: "10-10-2022",
14       }
15       const {
16         statusCode,
17         ok,
18         _body
19       } = await requester.post('/api/pets').send(petMock)
20       console.log(statusCode);
21       console.log(ok);
22       console.log(_body);
23     })
24   })
25 })
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

> mocha test/supertest.test.js

```
Testing Adoptme
  Test de mascotas
    200
    true
    {
      status: 'success',
      payload: {
        name: 'Patitas',
        especie: 'Pez',
```

# 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 necesitamos.

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!

# Elementos avanzados de Testing

# 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: 'correomau@correo.com',  
    password: '123'  
  }  
  
  //Ahora, Obtendremos 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 deestructurar 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('correomau@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:

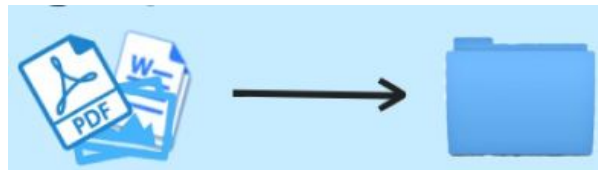
- ✓ Que el endpoint de *unprotectedLogin* devuelva una cookie de nombre ***unprotectedCookie***.
- ✓ Que el endpoint *unprotectedCurrent* devuelva al usuario completo, evaluar que se encuentren todos los campos que se guardaron en la base de datos.

# Testing con upload de archivos

# 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. 🚀**



# Recordemos que no enviamos un json 🙄🙄

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",  
      especie: "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.

```
//Corroboramos que la petición haya resultado en OK
expect(result.status).to.be.eql(200);
//Corroboramos 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 envían 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 deestructurar éste correctamente

### Test uploads

- ✓ Debe poder crearse una mascota con la ruta de la imagen (85ms)

6 passing (610ms)

¿Preguntas?



# Módulos de Testing para proyecto final



# 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 incluir por lo menos 3 tests desarrollados para
  - Router de products.
  - Router de carts.
  - Router de sessions.
- ✓ 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.**

# Resumen de la clase hoy

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



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

**#DemocratizandoLaEducación**