



Clase 42. Programación Backend

Testeo de funcionalidades



OBJETIVOS DE LA CLASE

- Realizar solicitudes HTTP al servidor en Node.
- Testear funcionalidades en Node con librerías Mocha, Supertest y Chai.

CRONOGRAMA DEL CURSO

Clase 41



**Desarrollo de un servidor
web basado en capas**

Clase 42



**Testeo de
funcionalidades**

Clase 43



**Desarrollo de un
servidor web basado en
capas - Parte 2**

CLIENTES HTTP



¿De qué se trata?

- El cliente HTTP es el encargado abrir una sesión HTTP y de enviar la solicitud de conexión al servidor.
- Hay varias formas de realizar solicitudes HTTP en Node a través de **clientes HTTP**. Existen dos tipos principales de clientes.
 - **Internos:** módulo HTTP o HTTPS estándar que vienen en la librería estándar de Node.
 - **Externos:** paquetes de NPM (como Axios o Got, entre otros) instalables como cualquier módulo.

Veámoslos...

CLIENTES HTTP
INTERNOS



Utilizar Módulo HTTP(S)

Ejemplo
en vivo



```
const https = require('https')
const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.end()
```

- Para usar el módulo de HTTP(S) de Node, simplemente lo requerimos como hacemos siempre.
- Podemos hacer una petición por GET, usando un código similar al mostrado de ejemplo.
- Usamos el método `http.request` para hacer la petición al servidor Node. Le pasamos las *options* como primer parámetro y un *callback* como segundo.



Utilizar Módulo HTTP(S)

Ejemplo
en vivo



- Para hacer una petición por POST, podemos usar un código como el que mostramos.
- Es muy similar al de GET. En el *options* agregamos los *headers*. Y luego, usamos el método `req.write` para poder guardar la data.
- Para peticiones PUT y DELETE, el formato es el mismo, solo que cambiamos *options.method* según corresponda.

```
const https = require('https')

const data = JSON.stringify({
  todo: 'Buy the milk'
})

const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length
  }
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.write(data)
req.end()
```

CODER HOUSE



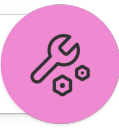
PETICIÓN CON HTTP

Tiempo: 10 minutos

PETICIÓN CON HTTP

Tiempo: 10 minutos

Desafío
generico



1. Realizar un pedido de recursos que se encuentran en la URL:
<https://jsonplaceholder.typicode.com/posts>. Para ello utilizar el módulo http nativo de node.js (options -> port: 80).
→ Estos recursos vienen dentro de un array de objetos. Al recibirlos, almacenarlos en un archivo llamado postsHttp.json conservando su estructura (respetar tabuladores, saltos de línea, etc.).
2. Realizar la misma solicitud, pero esta vez usando el módulo https interno de node.js. El archivo en el cual se guardará la respuesta será postsHttps.json (options -> port: 443).

CLIENTES HTTP
EXTERNOS

AXIOS



- Axios es una biblioteca de solicitudes muy popular basada en promesas.
 - Es un cliente HTTP disponible tanto para el navegador como para Node.
 - Incluye también funciones útiles como interceptar datos de solicitud y respuesta, y la capacidad de transformar automáticamente los datos de solicitud y respuesta a JSON.
- Lo empezamos a usar instalando el módulo con el comando

```
$ npm install axios
```



AXIOS - Petición GET

Ejemplo
en vivo



A

```
const axios = require('axios');

// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .then(function () {
    // always executed
  });

// Optionally the request above could also be done as
axios.get('/user', {
  params: {
    ID: 12345
  }
})
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  })
  .then(function () {
    // always executed
  });
```

- Para realizar una petición por GET al servidor usando Axios, podemos usar un código similar al mostrado. Tenemos dos opciones:

A. Definir el ID como query en la url.

B. Definir en la propiedad params.

- Además, al funcionar con promesas, podemos usar la sintaxis de Async/Await (**C**)

B

```
// Want to use async/await? Add the `async` keyword to your outer function/method.
async function getUser() {
  try {
    const response = await axios.get('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```

C

CODER HOUSE



AXIOS - Petición GET

Ejemplo
en vivo



```
axios.post('/user', {  
  firstName: 'Fred',  
  lastName: 'Flintstone'  
})  
.then(function (response) {  
  console.log(response);  
})  
.catch(function (error) {  
  console.log(error);  
});
```

- Para hacer una petición por POST al servidor, usando Axios podemos usar un código similar al que se muestra.
- A diferencia del de GET, pasamos como segundo parámetro del método `axios.post` un objeto con la data que se quiere guardar en la petición.



AXIOS - Múltiples peticiones

Ejemplo
en vivo



```
function getUserAccount() {  
  return axios.get('/user/12345');  
}  
  
function getUserPermissions() {  
  return axios.get('/user/12345/permissions');  
}  
  
Promise.all([getUserAccount(), getUserPermissions()])  
  .then(function (results) {  
    const acct = results[0];  
    const perm = results[1];  
  });
```

- Si necesitamos hacer múltiples peticiones podemos usar un código como este.
- Vemos que hacemos por separado las peticiones, y luego con *Promise.all* las ejecutamos.

GOT



- Got es otro módulo para peticiones HTTP popular para Node que afirma ser una "biblioteca de peticiones HTTP potente y amigable para los humanos para Node."
- Cuenta con una API basada en promesas, y compatibilidad con HTTP/2 y su API de paginación son las PVU de Got.
- Actualmente, Got es módulo cliente HTTP más popular para Node.
- Instalamos Got con el comando `$ npm install got`

GOT - Petición GET

Ejemplo
en vivo



```
const got = require('got');

(async () => {
  try {
    const response = await got('https://sindresorhus.com');
    console.log(response.body);
    //=> '<!doctype html> ...'
  } catch (error) {
    console.log(error.response.body);
    //=> 'Internal server error ...'
  }
})();
```

- Para hacer una petición por GET podemos usar un código como este.
- En `got` el parámetro es la url.
- El `response.body` es el HTML de la url especificada.

GOT - Petición POST

Ejemplo
en vivo

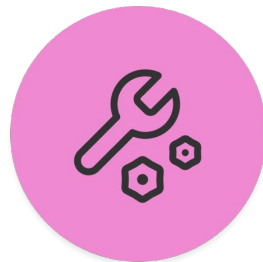


- En este caso, debemos especificar que usamos el método `got.post`.
- Como primer parámetro pasamos la url, y como segundo un objeto, especificando la data (propiedad `json`) y el `responseType` indicando que la respuesta es en formato `json`.

```
const got = require('got');

(async () => {
  const {body} = await got.post('https://httpbin.org/anything', {
    json: {
      hello: 'world'
    },
    responseType: 'json'
  });

  console.log(body.data);
  //=> {hello: 'world'}
})();
```



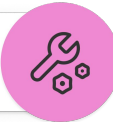
SERVIDOR CON PETICIONES HTTP

Tiempo: 10 minutos

SERVIDOR CON PETICIONES HTTP

Tiempo: 10 minutos

Desafío
generico



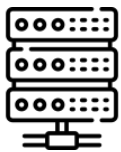
1. Realizar un servidor simple, que utilice el módulo nativo http de node.js para responder un objeto con la fecha y hora actual, ante un request hacia su ruta raíz. Ej. de respuesta: { FyH: '1/6/2021 15:14:21' }.
- Mediante un cliente http nativo de node, pedir la fecha y hora al servidor realizado y representarla en consola en formato JSON.
2. Realizar la misma operación utilizando Axios y Got.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

TEST DE SERVIDORES



¿De qué se trata?



- Los test son una parte fundamental del desarrollo de software. Hay diferentes prácticas como TDD, BDD y aparte diferentes tipos de test como test de aceptación, test de seguridad etc.
- Indistintamente de las prácticas, nombres y demás, cuando desarrollamos una API queremos que se comporte como debe cuando se realizan peticiones.
- Por ejemplo si realizamos una petición a un endpoint que no existe debería devolvernos un 404 como código de respuesta. Si hacemos una petición por post para crear un recurso debe devolvernos un 201 y un *header location* con la url donde se puede acceder al recurso creado. Que esto funcione de esta forma lo debemos testear previo al funcionamiento real de la aplicación.

TDD



TDD: Test Driven Development

TDD o Desarrollo guiado por pruebas es una técnica de programación que se centra en el hecho de que los test los escribimos **antes de programar la funcionalidad**, siguiendo el ciclo falla, pasa, refactoriza [red, green, refactor] intentando así mejorar la calidad del software que producimos.



TDD: Test Driven Development

- Algunas ventajas de desarrollar de esta forma son:
 - Nos ayuda a pensar en cómo queremos desarrollar la funcionalidad.
 - Permite hacer software más modular y flexible.
 - Minimiza la necesidad de un «debugger».
 - Aumenta la confianza del desarrollador a la hora de introducir cambios en la aplicación.



TDD: Test Driven Development



- Algunos de los problemas que tiene TDD son:
 - Dificultades a la hora de probar situaciones en las que son necesarios test funcionales o de integración, como pueden ser Bases de Datos o Interfaces de Usuario.
 - A veces se crean test innecesarios que provocan una falsa sensación de seguridad, cuando en realidad no están probando más que el hecho de que un método haga lo que dice que hace.
 - Los test también hay que mantenerlos a la vez que se mantiene el código, lo cual genera un trabajo extra.



(...)



- Es difícil introducir TDD en proyectos que no han sido desarrollados desde el principio con TDD.
 - Para que sea realmente efectiva hace falta que todo el equipo de desarrollo haga TDD.
 - A veces el desarrollador se centra más en cómo construir una funcionalidad que en preguntarse si la funcionalidad es de verdad necesaria para el usuario o es como la quería el usuario.
- Para intentar solucionar algunos de estos problemas nace BDD y se usan otras técnicas como los Test de Aceptación.

BDD



BDD: Behaviour Driven Development



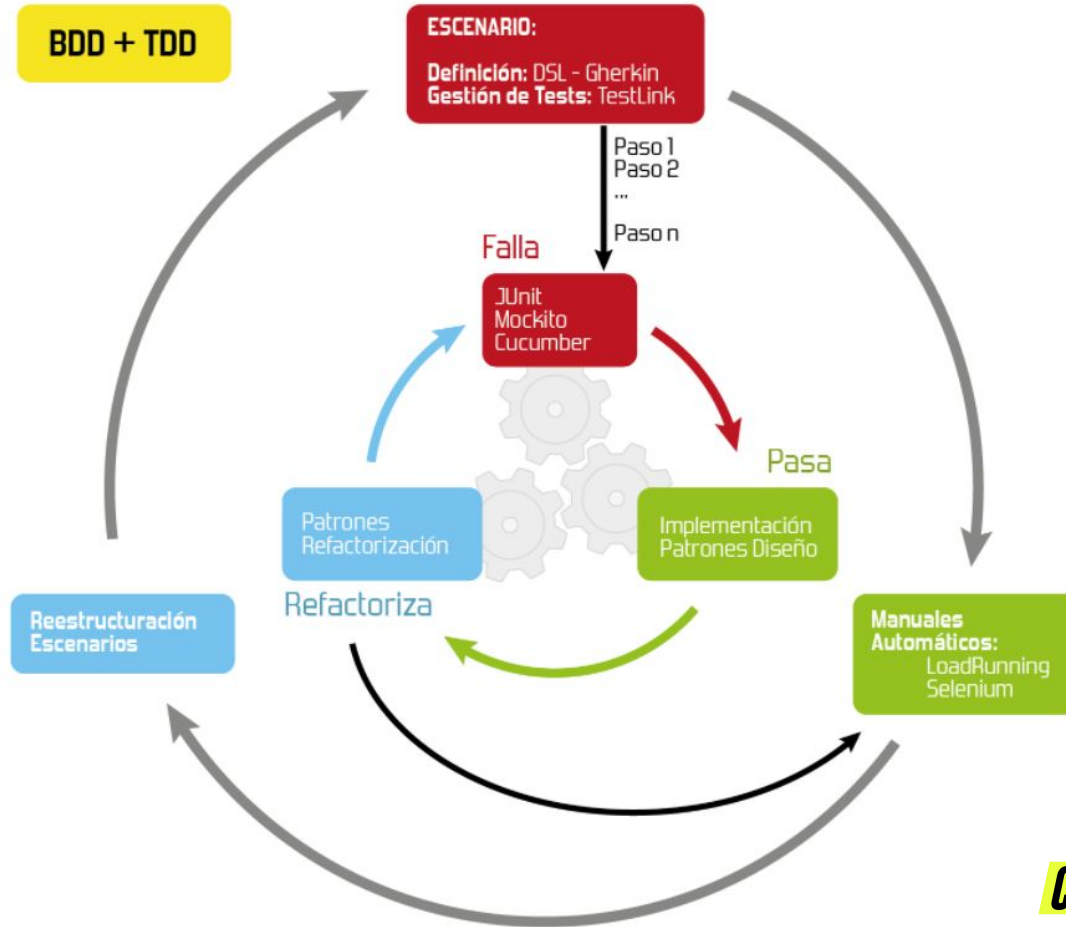
- El Desarrollo Guiado por el Comportamiento (BDD) es un proceso que amplía las ideas de TDD y las combina con otras ideas de diseño de software y análisis de negocio para proporcionar un proceso a los desarrolladores, con la intención de mejorar el desarrollo del software.
- BDD se basa en TDD formalizando las mejores prácticas de TDD, clarificando cuáles son y haciendo énfasis en ellas.
- En BDD no probamos solo unidades o clases, probamos escenarios y el comportamiento de las clases a la hora de cumplir dichos escenarios, los cuales pueden estar compuestos de varias clases.



BDD



COMPORTAMIENTO





BDD: Behaviour Driven Development



- Ventajas que añade BDD a las de TDD:



- Nos ayuda a centrarnos en lo que es verdaderamente importante para el 'negocio'.
- Si generamos las pruebas con un lenguaje concreto, nos pueden servir a la hora de hacer los test de Aceptación.

→ A la hora de llevar a la práctica BDD es muy recomendable usar un DSL (Domain Specific Language), es decir un lenguaje común sobre el que poder hacer los test y así disminuir la fricción a la hora de compartir los test.

MOCHA



¿De qué se trata?



- Mocha es un framework de pruebas para JavaScript que se ejecuta en Node y nos ayuda a tener un marco de trabajo para realizar nuestras pruebas de manera ordenada. Además se encarga de ejecutar los casos de prueba.
- Se utiliza para realizar pruebas unitarias o TDD. Sin embargo, no verifica el comportamiento de nuestro código. Entonces, para comparar los valores en una prueba, podemos usar el módulo **assert** de Node.
- Entonces, usamos Mocha como creador del plan de pruebas y assert como implementador de las mismas.



Creando test con Mocha



1. Primero creamos nuestro proyecto Node.
2. Luego, instalamos el módulo Mocha como dependencia de desarrollo (dev) ya que no lo vamos a necesitar en un entorno de producción.
3. Para instalarlo usamos el comando: ***npm i -D mocha***
4. El proyecto de ejemplo será un módulo llamado TODOS que administra una lista de elementos.



Creando test con Mocha

Ejemplo
en vivo



- El método *list* trae un listado de todos los elementos.
- El método *add* agrega un nuevo elemento.
- El método *complete* marca como completado un elemento.
- Los métodos de save lo guardan en un archivo, uno mediante *callback* y el otro con *promises*.

```
class Todos {  
  
  constructor() {  
    this.todos = []  
  }  
  
  list() {  
    return this.todos  
  }  
  
  add(title) {  
    let todo = {  
      title: title,  
      complete: false  
    }  
    this.todos.push(todo)  
  }  
  
  complete(title) {  
    if(this.todos.length === 0) {  
      throw new Error('No hay tareas')  
    }  
  
    let todoFound = false  
    this.todos.forEach( todo => {  
      if(todo.title === title) {  
        todo.complete = true  
        todoFound = true  
        return  
      }  
    })  
  
    if(!todoFound) {  
      throw new Error('Tarea no encontrada')  
    }  
  }  
}
```

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



- Los métodos de save lo guardan en un archivo, uno mediante *callback* y el otro con *promises*.

```
saveToFileCb(cb) {  
  let fileContents = ''  
  this.todos.forEach( todo => {  
    fileContents += `${todo.title},${todo.complete}`  
  })  
  fs.writeFile('todos.txt', fileContents, cb)  
}  
  
saveToFilePromise() {  
  let fileContents = ''  
  this.todos.forEach( todo => {  
    fileContents += `${todo.title},${todo.complete}`  
  })  
  
  return fs.promises.writeFile('todos.txt', fileContents)  
}  
  
module.exports = Todos
```

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



```
const Todos = require('./index')

const todos = new Todos()
console.log(todos.list())

todos.add("run code")
console.log(todos.list())

todos.add("otra tarea")
console.log(todos.list())

todos.complete("run code")
console.log(todos.list())
```

1. Primero vamos a realizar **pruebas manuales**. Para eso, ejecutaremos las funciones de nuestro código y observaremos el resultado para garantizar que cumpla con nuestras expectativas.
2. Creamos otro archivo, requerimos el anterior, y creamos una nueva instancia del mismo (clase Todos).
3. Luego, ejecutamos los métodos creados.



Creando test con Mocha



- A medida que vamos agregando más funcionalidades, se hace complicado testear manualmente, y puede generar errores.
- Por esto se recomienda hacer pruebas automatizadas, para lo que usamos Mocha. Estas son pruebas con secuencias de comandos escritas como cualquier otro bloque de código.
- Ejecutamos nuestras funciones con entradas definidas e inspeccionamos sus efectos para garantizar que se comportan como esperamos.
- Cuando escribimos nuevas pruebas junto a las funciones, podemos verificar que el módulo completo aún funcione, todo sin necesidad de recordar la manera en que se usa cada función en cada ocasión.



Creando test con Mocha

Ejemplo
en vivo



- Para comenzar a testear con Mocha, creamos un archivo llamado ***index.test.js***.
- En el mismo, primero requerimos nuestro módulo TODOS y el módulo assert con su propiedad strict.
- La propiedad strict del módulo assert nos permitirá usar las pruebas de igualdad especiales que se recomiendan desde Node y son adecuadas para una buena protección futura, ya que tienen en cuenta la mayoría de los casos de uso.

```
const Todos = require('./index')  
const assert = require('assert').strict
```



Creando test con Mocha

Ejemplo
en vivo



- Las pruebas estructuradas de Mocha siguen la plantilla de código que vemos.
- La función `describe()` se usa para agrupar pruebas similares. No es necesario ejecutar las pruebas, pero agruparlas facilita el mantenimiento de nuestro código. Se recomienda agrupar las pruebas de una manera que nos permita actualizar fácilmente las que son similares.
- La función `it()` contiene nuestro código de prueba.
- Se intenta que los nombres de las pruebas sean lo más descriptivo posible de lo que hace la misma.

```
describe([String with Test Group Name], function() {  
  it([String with Test Name], function() {  
    [Test Code]  
  });  
});
```

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



```
describe("test de integración de tareas", function() {  
  it('debería crear el contenedor de tareas vacío', function() {  
    const todos = new Todos()  
    assert.strictEqual(todos.list().length, 0)  
  })  
  
  it('debería adicionar tareas correctamente', function() {  
    const todos = new Todos()  
  
    todos.add("run code")  
    assert.strictEqual(todos.list().length, 1)  
    assert.deepStrictEqual(todos.list(), [ { title: 'run code', complete: false } ])  
  
    todos.add("otra tarea")  
    assert.strictEqual(todos.list().length, 2)  
    assert.deepStrictEqual(todos.list(), [  
      { title: 'run code', complete: false },  
      { title: 'otra tarea', complete: false }  
    ])  
  })  
  
  it('debería marcar una tarea como completa', function() {  
    const todos = new Todos()  
  
    todos.add("run code")  
    todos.add("otra tarea")  
  
    todos.complete("run code")  
    assert.deepStrictEqual(todos.list(), [  
      { title: 'run code', complete: true },  
      { title: 'otra tarea', complete: false }  
    ])  
  })  
})
```

- En el primer test, creamos una nueva instancia de *Todos* y verificamos que no tenga elementos.
- Desde el módulo *assert* usamos el método *notStrictEqual()*. Esta función toma dos parámetros: el valor que deseamos probar (llamado *actual*) y el valor que esperamos obtener (llamado *expected*). Si ambos argumentos son iguales, *notStrictEqual()* genera un error para que la prueba falle.

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



```
describe("test de integración de tareas", function() {  
  
  it('debería crear el contenedor de tareas vacío', function() {  
    const todos = new Todos()  
    assert.strictEqual(todos.list().length, 0)  
  })  
  
  it('debería adicionar tareas correctamente', function() {  
    const todos = new Todos()  
  
    todos.add("run code")  
    assert.strictEqual(todos.list().length, 1)  
    assert.deepStrictEqual(todos.list(), [ { title: 'run code', complete: false } ])  
  
    todos.add("otra tarea")  
    assert.strictEqual(todos.list().length, 2)  
    assert.deepStrictEqual(todos.list(), [  
      { title: 'run code', complete: false },  
      { title: 'otra tarea', complete: false }  
    ])  
  })  
  
  it('debería marcar una tarea como completa', function() {  
    const todos = new Todos()  
  
    todos.add("run code")  
    todos.add("otra tarea")  
  
    todos.complete("run code")  
    assert.deepStrictEqual(todos.list(), [  
      { title: 'run code', complete: true },  
      { title: 'otra tarea', complete: false }  
    ])  
  })  
  
})
```

- Ahora testeamos el método de agregar un nuevo elemento.
- Creamos una instancia de Todos, y le agregamos un nuevo elemento.
- Luego verificamos que la longitud del listado de los elementos sea 1 con `strictEqual()`.
- La función `deepStrictEqual()` prueba de forma recursiva que nuestros objetos esperados y reales tienen las mismas propiedades. En este caso, prueba que el array de elementos tenga las propiedades del que agregamos.

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



```
describe("test de integración de tareas", function() {

  it('debería crear el contenedor de tareas vacío', function() {
    const todos = new Todos()
    assert.strictEqual(todos.list().length, 0)
  })

  it('debería adicionar tareas correctamente', function() {
    const todos = new Todos()

    todos.add("run code")
    assert.strictEqual(todos.list().length, 1)
    assert.deepStrictEqual(todos.list(), [ { title: 'run code', complete: false } ])

    todos.add("otra tarea")
    assert.strictEqual(todos.list().length, 2)
    assert.deepStrictEqual(todos.list(), [
      { title: 'run code', complete: false },
      { title: 'otra tarea', complete: false }
    ])
  })

  it('debería marcar una tarea como completa', function() {
    const todos = new Todos()

    todos.add("run code")
    todos.add("otra tarea")

    todos.complete("run code")
    assert.deepStrictEqual(todos.list(), [
      { title: 'run code', complete: true },
      { title: 'otra tarea', complete: false }
    ])
  })
})
```

- En este tercer test, creamos una instancia de *Todos* y agregamos dos elementos.
- Luego, completamos uno de los dos.
- Finalmente, con `deepStrictEqual()` verificamos que las propiedades sean las especificadas para ambos elementos.
- Tiene que verificar que el elemento que completamos tenga en *true* la propiedad *complete*.



Creando test con Mocha

Ejemplo
en vivo



```
describe("comprobar error en completar tarea inexistente", function() {  
  
  it('deberia dar error cuando no hay tareas cargadas', function() {  
    const todos = new Todos()  
  
    const errorEsperado = new Error('No hay tareas')  
    assert.throws(() => {  
      todos.complete('una tareas más')  
    }, errorEsperado)  
  })  
  
  it('deberia dar error cuando la tarea a completar no existe', function() {  
    const todos = new Todos()  
    todos.add("run code")  
  
    const errorEsperado = new Error('Tarea no encontrada')  
    assert.throws(() => {  
      todos.complete('una tareas más')  
    }, errorEsperado)  
  })  
})
```

- En este caso, verificamos los errores.
- En el primer test, verificamos que nos de el error “no hay tareas” cuando se completa una tarea, y en el listado no había ninguna.
- En el segundo test, agregamos una tarea y verificamos que nos de el error “Tarea no encontrada” cuando se completa una tarea que no existe en el listado.



Creando test con Mocha

Ejemplo
en vivo



- En este caso, realizamos el test sobre el método `saveToFileCb()` para verificar su correcto funcionamiento. Creamos una instancia de `Todos`, agregamos un elemento, y verificamos que se guarde correctamente en el archivo utilizando este método.
- Como el método a probar es asíncrono con *callback*, usamos el parámetro `done` como argumento del callback de la función `it()`. Mocha utiliza la función de *callback* `done()` para indicarle cuando se completa una función asíncrona.

→ Vemos que toda la prueba para esta función reside en el callback de la misma. Sino fuera así, fallaría al ejecutarse antes de completar la escritura del archivo.

```
describe("comprobando que saveToFileCb() funcione bien", function() {  
  it('debería guardar una tarea en el archivo todos.txt', function(done) {  
    const todos = new Todos()  
    todos.add('guardar tarea callback')  
    todos.saveToFileCb( err => {  
      assert.strictEqual(fs.existsSync('todos.txt'), true)  
      let contenidoEsperado = 'guardar tarea callback,false'  
      let content = fs.readFileSync('todos.txt').toString();  
      assert.strictEqual(content, contenidoEsperado)  
      done(err)  
    })  
  })  
})
```

CODER HOUSE



Creando test con Mocha



- Para mejorar casos de prueba, podemos usar algunos de estos enlaces:
 - ***before***: este enlace se ejecuta una vez antes de que se inicie la primera prueba.
 - ***beforeEach***: este se ejecuta antes de cada caso de prueba.
 - ***after***: este se ejecuta una vez después que se completa el último caso de prueba.
 - ***afterEach***: este se ejecuta después de cada caso de prueba.
- Cuando probamos una función o característica varias veces, los enlaces son útiles, ya que nos permiten separar el código de configuración de la prueba (como la creación del objeto *Todos*) desde el código de aserción de esta.



Creando test con Mocha

Ejemplo
en vivo



```
describe("comprobando que saveToFilePromises() funcione bien", function() {  
  
  before(function() {  
    console.log('\n***** Comienzo TOTAL de Test *****')  
  })  
  
  after(function() {  
    console.log('\n***** Fin TOTAL de Test *****')  
  })  
  
  beforeEach(function() {  
    console.log('\n***** Comienzo Test *****')  
  })  
  
  beforeEach(function() {  
    this.todos = new Todos()  
  })  
  
  afterEach(function() {  
    if(fs.existsSync('todos.txt')) {  
      fs.unlinkSync('todos.txt')  
    }  
  })  
  
  afterEach(function() {  
    console.log('***** Fin Test *****\n')  
  })  
})
```

- Vamos a testear el método `saveToFilePromises()`.
- Para esto, vemos que podemos usar los enlaces.
- En este caso, usamos *before* y *after* solamente para loguear el comienzo y final del test.
- Con *beforeEach* creamos la instancia de Todos.
- Con *afterEach* guardamos los elementos en el archivo.

CODER HOUSE



Creando test con Mocha

Ejemplo
en vivo



```
it('debería guardar una tarea en el archivo todos.txt (then/catch)', function() {  
  //const todos = new Todos()  
  //todos.add('guardar tarea Promises TC')  
  this.todos.add('guardar tarea Promises TC')  
  //return todos.saveToFilePromise().then(() => {  
  return this.todos.saveToFilePromise().then(() => {  
    assert.strictEqual(fs.existsSync('todos.txt'), true)  
    let contenidoEsperado = 'guardar tarea Promises TC,false'  
    let content = fs.readFileSync('todos.txt').toString();  
    assert.strictEqual(content, contenidoEsperado)  
  })  
})  
  
it('debería guardar una tarea en el archivo todos.txt (async/await)', async function() {  
  //const todos = new Todos()  
  //todos.add('guardar tarea Promises AA')  
  this.todos.add('guardar tarea Promises AA')  
  
  //await todos.saveToFilePromise()  
  await this.todos.saveToFilePromise()  
  
  assert.strictEqual(fs.existsSync('todos.txt'), true)  
  let contenidoEsperado = 'guardar tarea Promises AA,false'  
  let content = fs.readFileSync('todos.txt').toString();  
  assert.strictEqual(content, contenidoEsperado)  
})  
})
```

- Realizamos entonces los test de este método.
- En el primer caso, lo hacemos usando **then/catch**. En esta, la prueba debe estar dentro del `.then()`.
- En el segundo caso, lo hacemos con **async/await**. El *callback* de la función `it()` debe ser *async*, y la ejecución del método a testear debe ser con *await*. De esta forma, se espera a que guarde los datos en el archivo para ejecutar el test.



Ejecutando los test creados

Ejemplo
en vivo



- Para ejecutar los test, en primer lugar, debemos ir a nuestro archivo Package.json y modificar el objeto *scripts* como vemos:

```
"scripts": {  
  "test": "mocha index.test.js",  
  "test-manual": "node index.prueba.manual.js"  
},
```

- Ejecutamos los test manuales que hicimos con el comando: `$ npm run test-manual`

- Obtenemos en la consola:

```
> node index.prueba.manual.js  
  
[  
  [ { title: 'run code', complete: false } ]  
  [  
    { title: 'run code', complete: false },  
    { title: 'otra tarea', complete: false }  
  ]  
  [  
    { title: 'run code', complete: true },  
    { title: 'otra tarea', complete: false }  
  ]  
]
```



Ejecutando los test creados

Ejemplo
en vivo



- Ejecutamos ahora los test realizados con Mocha y assert con el comando:

```
$ npm test
```

ó

```
$ npm run test
```

- Obtenemos en la consola:

```
> mocha index.test.js

test de integración de tareas
  ✓ debería crear el contenedor de tareas vacío
  ✓ debería adicionar tareas correctamente
  ✓ debería marcar una tarea como completa

comprobar error en completar tarea inexistente
  ✓ debería dar error cuando no hay tareas cargadas
  ✓ debería dar error cuando la tarea a completar no existe

comprobando que saveToFileCb() funcione bien
  ✓ debería guardar una tarea en el archivo todos.txt

comprobando que saveToFilePromises() funcione bien

***** Comienzo TOTAL de Test *****

***** Comienzo Test *****
  ✓ debería guardar una tarea en el archivo todos.txt (then/catch)
***** Fin Test *****

***** Comienzo Test *****
  ✓ debería guardar una tarea en el archivo todos.txt (async/await)
***** Fin Test *****

***** Fin TOTAL de Test *****

8 passing (47ms)
```

CODER HOUSE

MOCHA CON CHAI Y SUPERTEST



SuperTest



- SuperTest es una librería de Node que proporciona una abstracción de alto nivel para probar solicitudes HTTP, perfecto para API.
- Si tenemos una aplicación Node que ejecuta un servidor HTTP (como una aplicación Express), podemos realizar solicitudes usando SuperTest directamente sin necesidad de un servidor en ejecución.
- Una de las cosas buenas de SuperTest es que, si bien puede ejecutar pruebas sin herramientas adicionales, puede integrarse muy bien con otros marcos de prueba, como veremos a continuación.



Chai



- Chai es una librería de assertions que se puede emparejar con otros marcos de prueba como Mocha.
- Si bien no es estrictamente necesario para escribir un conjunto de pruebas, proporciona un estilo más expresivo y legible para nuestras pruebas.
- Al igual que Mocha, Chai nos permite elegir aserciones de estilo BDD (esperar) o estilo TDD (afirmar) para que podamos combinar la librería con la mayoría de los frameworks sin ningún conflicto.



Construyendo un proyecto

Ejemplo
en vivo



- Para comenzar a testear usando Mocha integrado con SuperTest y Chai, creamos un proyecto de Node.
- Vamos a instalar estas 3 dependencias en Dev:
npm i -D mocha supertest chai (y nodemon si no la tienen global)
- Vamos a crear una API REST con algunas operaciones de CRUD sobre usuarios. Guardaremos los datos en una base de datos de MongoDB usando Mongoose.
- Utilizaremos una estructura de proyecto similar a la que venimos proponiendo en estas últimas clases.



Construyendo un proyecto

Ejemplo
en vivo



```
import express from 'express'

import RouterUsuarios from '../rutas/usuarios.js'

const app = express()

app.use(express.urlencoded({ extended: true }))
app.use(express.json())

app.use('/api/usuarios', new RouterUsuarios())

export default app
```

Nuestro punto de inicio es un servidor Express en el archivo *server.js*. Luego de configurarlo con los middlewares y el router correspondientes, lo exportamos para utilizarlo desde donde lo deseemos.



Construyendo un proyecto

Ejemplo
en vivo



```
import { Router } from 'express'
import { getController, postController } from '../controladores/index.js'

const router = Router()

router.post('/', (req, res) => postController.execute(req, res))
router.get('/:id?', (req, res) => getController.execute(req, res))

class RouterUsuarios {
  constructor() {
    return router
  }
}

export default RouterUsuarios
```

```
import GetController from './GetController.js'
import PostController from './PostController.js'

const getController = new GetController()
const postController = new PostController()

export { getController, postController }
```

Nuestro único router se encarga de delegar las peticiones que lleguen a la ruta de la api de usuarios con métodos get y post, a los controladores correspondientes, traídos desde un factory de controladores



Construyendo un proyecto

Ejemplo
en vivo



```
import ApiUsuarios from '../api/usuarios.js'

export default class GetController {
  constructor () {
    this.api = new ApiUsuarios ()
  }

  async execute (req, res) {
    try {
      const result = await
this.api.get (req.params.id)
      res.json (result)
    } catch (error) {
      res.send (error)
    }
  }
}
```

```
import ApiUsuarios from '../api/usuarios.js'

export default class PostController {
  constructor () {
    this.api = new ApiUsuarios ()
  }

  async execute (req, res) {
    try {
      const result = await
this.api.post (req.body)
      res.json (result)
    } catch (error) {
      res.send (error)
    }
  }
}
```

Ambos controladores son similares, y consumen los servicios de la Api de Usuarios, respondiendo acordemente con el formato requerido por el protocolo HTTP.



Construyendo un proyecto

Ejemplo
en vivo



```
import ApiUsuarios from '../api/usuarios.js'

export default class GetController {
  constructor () {
    this.api = new ApiUsuarios ()
  }

  async execute (req, res) {
    try {
      const result = await
this.api.get (req.params.id)
      res.json (result)
    } catch (error) {
      res.send (error)
    }
  }
}
```

```
import ApiUsuarios from '../api/usuarios.js'

export default class PostController {
  constructor () {
    this.api = new ApiUsuarios ()
  }

  async execute (req, res) {
    try {
      const result = await
this.api.post (req.body)
      res.json (result)
    } catch (error) {
      res.send (error)
    }
  }
}
```

Ambos controladores son similares, y consumen los servicios de la Api de Usuarios, respondiendo acordemente con el formato requerido por el protocolo HTTP.



Construyendo un proyecto

Ejemplo
en vivo



```
import { validarSchema } from '../validaciones/index.js'
import { daoUsuarios } from '../daos/index.js'

export default class ApiUsuarios {
  constructor () {
    this.usuariosDao = daoUsuarios
  }

  async post(usuario) {
    const { result, error } = validarUsuario(usuario)
    if (result) {
      try {
        const nuevoUsuario = { ...usuario, id: generarId() }
        await this.usuariosDao.create(nuevoUsuario)
        console.log('Usuario incorporado')
        return nuevoUsuario
      } catch (error) {
        new Error(`error en escritura de usuario: ${error}`)
      }
    } else {
      throw new Error(error)
    }
  }

  async get(query = {}) {
    try {
      const usuarios = await this.usuariosDao.find(query)
      return usuarios
    } catch (error) {
      throw new Error(`error en lectura de usuarios: ${err}`)
    }
  }
}
```

La api de usuarios utiliza un validador para verificar la validez del esquema de los datos recibidos desde el cliente, y un Dao de usuarios para el acceso a datos.



Construyendo un proyecto

Ejemplo
en vivo



```
import mongoose from 'mongoose'
import { jsSchema as usuarioSchema } from '../modelos/usuario.js'
const Schema = mongoose.Schema

const usuariosDao = mongoose.model('Usuario', new Schema(usuarioSchema))

class DaoUsuarios {
  constructor() {
    return usuariosDao
  }
}

export default DaoUsuarios
```

Nuestro Dao se conecta directamente con MongoDB a través del driver de mongoose, obteniendo el esquema del usuario desde su modelo.

```
export const jsSchema = {
  id: String,
  nombre: String,
  email: String
}
```



Construyendo un proyecto

Ejemplo
en vivo



```
export default class DaoUsuariosMem {
  constructor () {
    this.usuarios = []
  }

  static #matches (query, usuario) {
    for (const [ k, v ] of Object.entries(query)) {
      if (usuario[ k ] !== v) return false
    }
    return true
  }

  find(query) {
    return this.usuarios.filter(u => DaoUsuariosMem.#matches(query, u))
  }

  create(usuario) {
    this.usuarios.push(usuario)
  }
}
```

```
import DaoUsuariosMongoDb from './usuariosDaoMongoDb.js';
import DaoUsuariosMem from './usuariosDaoMem.js';

const persistencia = process.env.PERSISTENCIA || 'MEM'

let daoUsuarios
switch (persistencia) {
  case 'MONGO':
    daoUsuarios = new DaoUsuariosMongoDb()
    break
  default:
    daoUsuarios = new DaoUsuariosMem()
}

export { daoUsuarios }
```

Opcionalmente, contamos con un dao de persistencia en memoria, para ejecutar nuestras pruebas más rápidamente. Elegimos nuestro DAO vía opciones de configuración por línea de comando.



Creando los test


Ejemplo
en vivo



Vamos ahora entonces al archivo `apirestfull.test.js` donde vamos a escribir los test.

1. Primero requerimos el módulo `supertest` con la url como método. También la librería `chai` con el método `expect`. Y también requerimos un generador de usuarios.

 `apirestfull.test.js` ✕

test >  `apirestfull.test.js` > ...

```
1  const request = require('supertest')('http://localhost:8080')
2  const expect = require('chai').expect
3  const generador = require('../generador/usuarios')
```



Creando los test

Ejemplo
en vivo



JS usuarios.js X

generador > JS usuarios.js > ...

```
1  const faker = require('faker')
2
3  //faker.locale = 'es'
4
5  const get = () => ({
6    nombre: faker.name.firstName(),
7    email: faker.internet.email()
8  })
9
10 module.exports = {
11   get
12 }
```

2. En el archivo de generador de usuarios, usamos un módulo llamado faker. Este módulo nos permite crear un usuario random, falso, con las propiedades que le especificamos.

3. Instalamos el módulo con el comando:

```
$ npm install --save -dev faker
```

4.  Luego, el código del archivo queda como vemos en la imagen.



Creando los test

Ejemplo
en vivo



5. Volviendo al archivo de test, primero escribimos la prueba para la petición de usuarios por GET.
6. Primero, la prueba carga la librería SuperTest y la asigna a la solicitud de variable `(request.get())`.
7. Luego, con el **expect** de Chai, definimos la respuesta de qué se espera y qué va a ser. En este caso esperamos que el status de la respuesta sea igual a 200.

```
describe('test api rest full', () => {  
  describe('GET', () => {  
    it('debería retornar un status 200', async () => {  
      let response = await request.get('/api')  
      //console.log(response.status)  
      //console.log(response.body)  
      expect(response.status).toEqual(200)  
    })  
  })  
})
```

CODER HOUSE



Creando los test

Ejemplo
en vivo

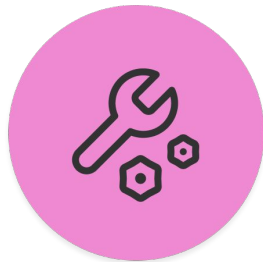


```
describe('POST', () => {
  it('debería incorporar un usuario', async () => {
    /* let usuario = {
      nombre: 'Pepe',
      email: 'pepe@gmail.com'
    } */
    let usuario = generador.get()

    let response = await request.post('/api').send(usuario)
    //console.log(response.status)
    //console.log(response.body)
    expect(response.status).to.eql(200)

    const user = response.body
    expect(user).to.include.keys('nombre','email')
    /* expect(user.nombre).to.eql('Pepe')
    expect(user.email).to.eql('pepe@gmail.com') */
    expect(user.nombre).to.eql(usuario.nombre)
    expect(user.email).to.eql(usuario.email)
  })
})
```

8. Finalmente, testeamos la petición por POST. En este caso, la data del body la generamos con el generador de usuarios.
9. Usamos ahora `request.post()` de `supertest` agregándole el body en el método `.send()`.
10. Verificamos que el status de la respuesta sea igual a 200.
11. Luego, verificamos que el body de la respuesta tenga las propiedades `nombre` y `email`. Y finalmente que los valores sean los agregados.



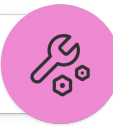
TESTEAR CON MOCHA

Tiempo: 10 minutos

TESTEAR CON MOCHA

Tiempo: 10 minutos

Desafío
generico



- Realizar un test de funcionamiento utilizando Mocha al servidor realizado en el desafío anterior.
- Realizar una suite de test que envíe al servidor 10 números consecutivos en su ruta '/ingreso' y luego comprobar en '/egreso' que esos números estén en cantidad y en orden.
- Integrar axios a la suite de test para realizar los request y utilizar sintaxis async await.
- Arrancar el servidor y luego el proceso de test.



TESTEAMOS NUESTRA API REST

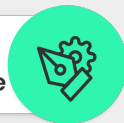
Retomemos nuestro trabajo para realizar test de algunas de las funcionalidades que tenemos en la API REST.

ESQUEMA API RESTful

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Desafío
entregable



>> Consigna:

Revisar en forma completa el proyecto entregable que venimos realizando, refactorizando y reformando todo lo necesario para llegar al esquema de servidor API RESTful en capas planteado en esta clase.

Asegurarse de dejar al servidor bien estructurado con su ruteo / controlador, negocio, validaciones, persistencia y configuraciones (preferentemente utilizando en la codificación clases de ECMAScript).

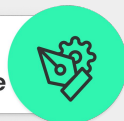
No hace falta realizar un cliente ya que utilizaremos tests para verificar el correcto funcionamiento de las funcionalidades desarrolladas.

TESTEAMOS NUESTRA API REST

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Desafío
entregable



>> Consigna (cont.):

- Desarrollar un cliente HTTP de pruebas que utilice Axios para enviar peticiones, y realizar un test de la funcionalidad hacia la API Rest de productos, verificando la correcta lectura de productos disponibles, incorporación de nuevos productos, modificación y borrado.
- Realizar el cliente en un módulo independiente y desde un código aparte generar las peticiones correspondientes, revisando los resultados desde la base de datos y en la respuesta del servidor obtenida en el cliente HTTP.
- Luego, realizar las mismas pruebas, a través de un código de test apropiado, que utilice mocha, chai y Supertest, para probar cada uno de los métodos HTTP de la API Rest de productos.
- Escribir una suite de test para verificar si las respuestas a la lectura, incorporación, modificación y borrado de productos son las apropiadas. Generar un reporte con los resultados obtenidos de la salida del test.

CODER HOUSE

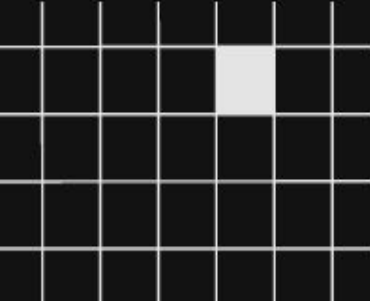
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Clientes HTTP internos y externos.
 - Axios - Got.
 - Test de servidores TDD y BDD.
 - Realizar test con Mocha, Supertest y Chai.
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE