



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



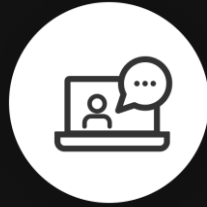
¿De qué se trata?



- 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



***¿Alguna pregunta hasta
ahora?***



PETICIÓN CON HTTP

Tiempo: 10 minutos

PETICIÓN CON HTTP

Desafío
generico



Tiempo: 10 minutos

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.).
1. 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).



***¿Alguna pregunta hasta
ahora?***

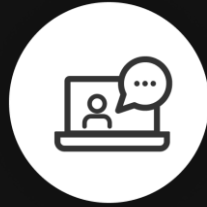
CLIENTES HTTP
EXTERNOS

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
```



***¿Alguna pregunta hasta
ahora?***

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`



***¿Alguna pregunta hasta
ahora?***



SERVIDOR CON PETICIONES HTTP

Tiempo: 10 minutos

SERVIDOR CON PETICIONES HTTP

Desafío
generico



Tiempo: 10 minutos

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: '2011-10-05T14:48:00.000Z' }.
- Mediante un cliente http nativo de node, pedir la fecha y hora al servidor realizado y representarla en consola en formato JSON.
1. Realizar la misma operación utilizando Axios y Got.



NÚMEROS ALEATORIOS CON CLIENTE HTTP EN AXIOS

Tiempo: 10 minutos

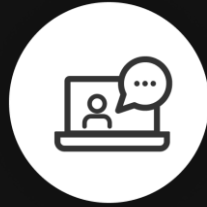
NÚMEROS ALEATORIOS CON CLIENTE HTTP EN AXIOS

Desafío
generico



Tiempo: 10 minutos

1. Desarrollar un servidor express en node.js que permita almacenar números aleatorios que ingresan desde la ruta post '/ingreso'. Dichos números persistirán en memoria.
2. Realizar un cliente http en axios que cada dos segundos envíe un número aleatorio al servidor en la ruta post '/ingreso' y otro cliente http en got que le pida a la ruta get '/egreso' cada 10 segundos la lista completa de números almacenados.



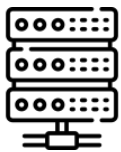
***¿Alguna pregunta hasta
ahora?***



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 a **CODER HOUSE**

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.

BDD



BDD: Behaviour Driven Development



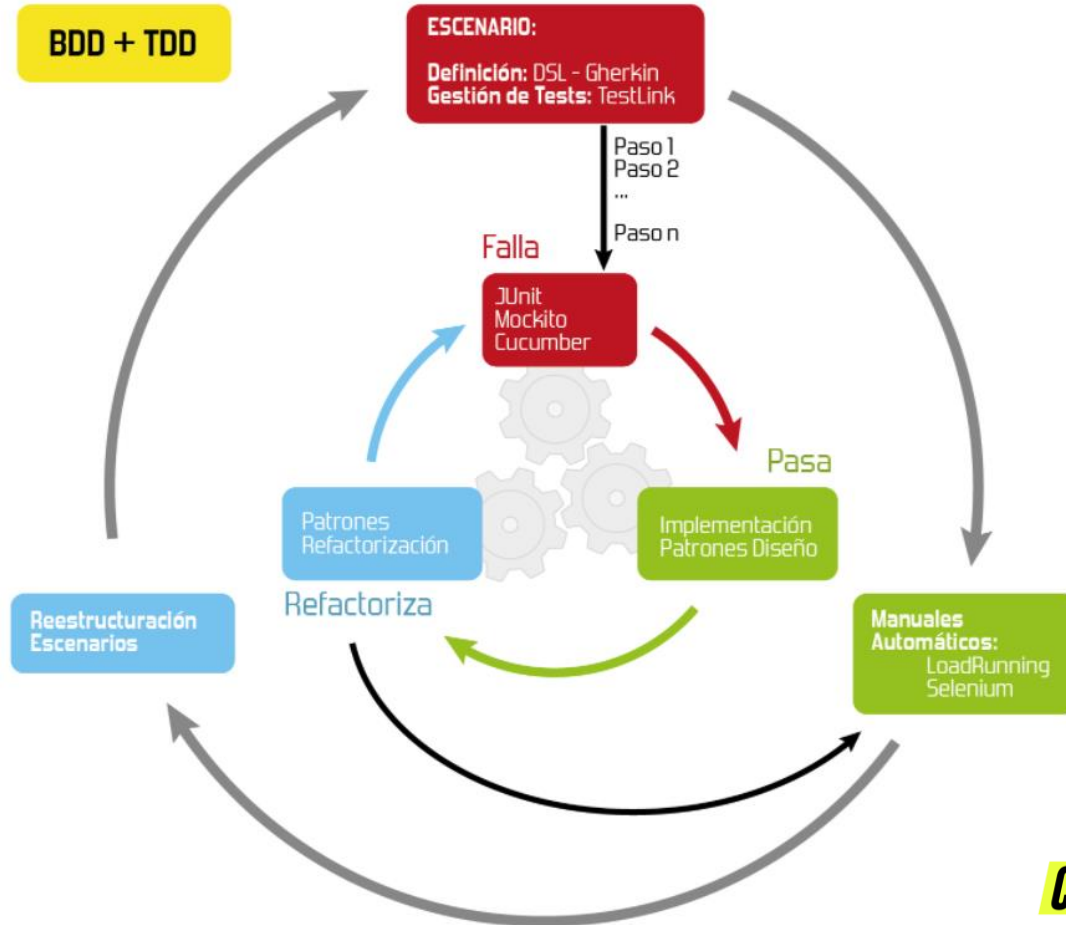
- El Desarrollo Guiado por el Comportamiento (BDD) es un proceso que amplía las ideas de TDD y las combina con otras ideas de diseño de software y análisis de negocio para proporcionar un proceso a los desarrolladores, con la intención de mejorar el desarrollo del software.
- BDD se basa en TDD formalizando 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.

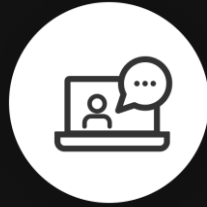


BDD



COMPORTAMIENTO





***¿Alguna pregunta hasta
ahora?***

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.



***¿Alguna pregunta hasta
ahora?***

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.



***¿Alguna pregunta hasta
ahora?***



TESTEAR CON MOCHA

Tiempo: 10 minutos

TESTEAR CON MOCHA

Desafío
generico



Tiempo: 10 minutos

- 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.

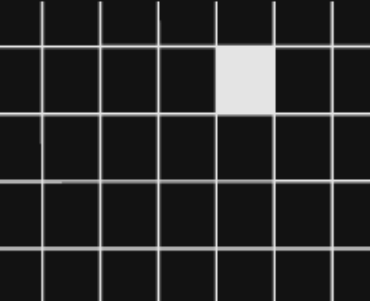
¿PREGUNTAS?





¡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