

## Qué es Node.js?

- Ejecute JavaScript en todas partes.
- Node.js es un entorno de ejecución de JavaScript multiplataforma, de código abierto y gratuito que permite a los desarrolladores escribir herramientas de línea de comandos y scripts del lado del servidor fuera de un navegador.

## Qué es Express?

- Es un Framework de Node.js o Java Script, que nos permite crear aplicaciones web del lado del servidor (Backend).
- Es el Framework más popular de Node.js
- Express es una herramienta de terceros, es código de otras personas. Muchas aplicaciones se complementan de módulos de terceros.
- Aquí en Express se utiliza Java Script puro.
- Para trabajar Express, se debe conocer Html, Css y JavaScript Basic
- npm es un manejador de módulos.
- ! Express es un módulo de npm!

node -v [Muestra la versión de node]

npm -v [Muestra la versión de npm]

## Proyecto node.js + Express + Mysql

1-Creamos una carpeta que va a hacer el nombre del proyecto Ejemplo  
Apirestnodejsmysql

2-Entramos a Visual Studio Code y nos ubicamos en la ruta del proyecto ó vamos a la ruta del proyecto y abrimos una ventana de Windows PowerShell y allí digitamos code . y me abre automáticamente mi proyecto con Visual Studio Code

3-Abrimos la terminal de comandos con Ctrl + ñ ó por el menú de opciones Terminal - New Terminal

4-Ejecutamos el siguiente comando npm init -y, npm init—yes, ó npm init, [Nos inicializa el proyecto para poder empezar a instalar las librerías, cualquiera de los tres es válido pero con el tercero te va a pedir por consola que le Agregues una Description y el Author.

Después de ejecutar el anterior comando se nos crea un archivo llamado package.json [Este archivo describe el proyecto] que contiene lo siguiente:

```
{  
  "name": "express",  
  "version": "1.0.0",
```

```

"description": "",
"main": "index.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC"
}

```

Por defecto se crea que la aplicación se inicia o ejecuta en el archivo `index.js`, pero se puede cambiar por el que se desee, preferiblemente dejarlo como `index.js`, que hay que crearlo para colocarle código.

## 5- Ahora instalamos las librerías que vamos a utilizar

Estas librerías se pueden ejecutar juntas simultáneamente separadas por espacio.

Ejemplo: `npm install express body-parser`

El install se puede reemplazar por i

`npm install express` [Me crea una carpeta `node_modules`, la cual me permite que express funcione, también me crea un archivo `package-lock.json`]

`npm install body-parser` [Para trabajar con peticiones post]

`npm install sequelize` [ORM de node.js para mysql, sql server, postgres, sqlite]

`npm install mysql` ó `npm install mysql2` [Cuando se utiliza `mysql2` es para utilizar la última versión, conecta el servidor con el banco de datos]

`express-myconnection` [Para la conexión de la base de datos]

`npm install express-promise-router`

`npm install express-validator`

`npm install bcryptjs` [Encripta el password de la base de datos]

`npm install cors`

```
npm install moment
```

```
npm install jwt-simple
```

```
npm install jsonwebtoken
```

```
npm install mongoose
```

```
npm install morgan
```

Cuando se crean o instalan las dependencias, automáticamente se actualiza el `package.json`, ejemplo instalé las siguientes dependencias y quedó así:

```
{
  "name": "apiRESTnodejsmysql",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
    "express-myconnection": "^1.0.4",
    "mysql": "^2.18.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

6- También hay unas dependencias de desarrollo

@babel/cli

@babel/core

@babel/node

@babel/preset-env

nodemon [Esta es la más importante]

Al momento de terminar con estas líneas de las librerías hay que colocar -D [esto indica que son dependencias de desarrollo]

npm i nodemon --save-dev [Esta línea de comando también es válida para instalar nodemon]

[ --save-dev esta línea me permite decirle que es una librería de desarrollo, es decir que sólo la necesito mientras programo]

[Esta dependencia me permite que el servidor se reinicie automáticamente cada vez que yo hago un cambio en mi aplicación]

Cuando se instala nodemon, ir a package.json y cambiar por index.js ó por el nombre de inicio que se hay decidido.

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

Por

```
"scripts": {  
  "start": "nodemon index.js"  
},
```

7- Luego que ya hayamos instalado las librerías o por lo menos las iniciales que necesitamos, porque podemos ir instalándolas a medida que las vamos requiriendo.

Creamos un archivo `index.js`, si no queremos que se llame así nuestro archivo de inicio, lo podemos cambiar, pero también hay que cambiarlo en `package.json`, que por defecto viene `index.js`. Ejemplo `"main": "index.js"`

Con `nodemon` para reiniciar el servidor se utiliza el siguiente comando: [ `node index.js` ]

`npm start` [Vuelve a reiniciar cada vez que se guarde algún archivo del proyecto]

`npm run dev` []

8- Escribimos el siguiente código en nuestro `index.js`

---

### EJEMPLO1

<https://www.youtube.com/watch?v=T6rGUZGAWBk>

```
const express = require('express');  
const bodyParser = require('body-parser'); [La que me permite enviar objetos asociados a una  
petición post ]  
const app = express();  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: true })); [Me codifica la url]  
app.get('/', (req, res) => {  
  res.send('Hola Mundo');  
});  
app.listen(3000, () => {  
  console.log('Servidor arrancado');  
});
```

Ahora me voy a la terminal de comandos y ejecuto: `node index.js`

me debe mostrar en pantalla: [Servidor arrancado] si no hay inconvenientes o errores

Luego verifico en el navegador, escribo `localhost:3000` y me debe mostrar [Hola Mundo]

---

Otras formas de comenzar con el `index.js` seria:

---

## EJEMPLO2

<https://www.youtube.com/watch?v=794Q71KVw1k>

Este código es de la creación de un servidor con Node.js

```
const http = require('http');  
  
const server = http.createServer((req, res) => {  
  res.status = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello world');  
});
```

```
server.listen(3000, () => {  
  console.log('Serve on Port 3000');  
});
```

Me voy a la terminal de comandos y ejecuto:

```
node index.js
```

Y me debería mostrar CASO 2: `Hello world`

En el navegador `localhost:3000`

---

## EJEMPLO3

<https://www.youtube.com/watch?v=yaMcqleO9BU&list=PL-yog8huBN1jkfGnJ82HMTzDx9g7tOU0N>

```
// const express = require('express');  
import express from 'express';  
import morgan from 'morgan';  
  
const app = express();  
app.set('port', process.env.PORT || 3000);  
  
app.listen (app.get('port'), ()=>{  
  console.log('server on port: ', app.get('port'))  
});
```

Me voy a la terminal de comandos y ejecuto:

```
node index.js
```

Y me debería mostrar CASO 3: `server on port: 3000`

En el navegador `localhost:3000`

---

#### EJEMPLO4

<https://www.youtube.com/watch?v=OWukxSRtr-A>

```
const express = require('express');  
const app = express();  
app.set('port', process.env.PORT || 9000)  
  
app.get('/', (req,res) =>{
```

```
res.send('Welcome to my API');  
});
```

```
app.listen(app.get('port'), => {  
  console.log('Server running on port', app.get('port'))  
});
```

Me voy a la terminal de comandos y ejecuto:

```
node index.js
```

Y me debería mostrar CASO 3: `Server running on port: 9000`

En el navegador `localhost:9000`

---

## Sequelize

```
npm install sequelize
```

 [Trabaja con Mysql, Postgre, Maria db, Sqlite, SqlServer]

El soporte de Node.js para bases de datos NoSQL es muy completo, pero si uno quiere convertirse en la plataforma de referencia se necesitan **soportar también bases de datos SQL**. En estos servidores el uso de frameworks como **Hibernate y Entity Framework** se ha convertido en algo habitual. [Sequelize](#) es el framework ORM de Node.js para soluciones como MySQL o Postgre.

Yo puedo abstraerme del lenguaje de Mysql y simplemente crear objetos, editarlos, borrarlos.

```
npm install sequelize
```

Instalado el framework el siguiente paso es instalar un soporte para una base de datos concreta, en esta caso MySQL.

```
npm install mysql
```

Vamos a utilizar el framework:

```
var Sequelize = require('Sequelize');
```

```
var sequelize = new Sequelize('genbetadev', 'miusuario', 'miclave', {
```

```
  host: 'localhost',
```



```

    dialect: 'mysql',

    pool: {
      max: 5,
      min: 0,
      idle: 10000
    },
  },
});

```

Instanciamos **Sequelize** y configuramos los típicos parámetros de usuario, clave etc. Al tratarse de un framework de persistencia \*\*es habitual definir el dialecto que se va a usar y el tamaño del pool de conexiones. En este caso optaremos por **MySQL**.

## Utilizando Sequelize

El siguiente paso será definir una entidad de dominio y persistirla contra la base de datos.

```

var Persona = sequelize.define('Persona', {
  nombre: {
    type: Sequelize.STRING,
    field: 'nombre'
  },
  edad: {
    type: Sequelize.INTEGER,
    field: 'edad'
  }
}, {
});

```

```
freezeTableName: true
```

```
});
```

Hemos elegido el concepto de\*\* Persona con nombre y edad\*\*. **Sequelize** hace uso de un API orientado a promesas para gestionar la persistencia.

```
var p1=Persona.sync({force: true}).then(function () {
```

```
return Persona.create({
```

```
  nombre: 'pepe',
```

```
  edad: 20
```

```
});
```

```
});
```

---

Creamos un archivo llamado bd.js

Aquí configuramos la conexión a la base de datos.

```
https://www.youtube.com/watch?v=T6rGUZGAWBk
```

```
const Sequelize = require('sequelize');
```

```
const ClientesModel = require('./models/clientes');
```

```
const sequelize = new Sequelize('AAAAAA ', 'BBBBBBB ', 'CCCCCCC', {
```

```
  host: 'www.sigp.com.co',
```

```
  dialect: 'mysql'
```

```
});
```

```
const Clientes = ClientesModel(sequelize, Sequelize);
```

```
sequelize.sync({ force: false })

    .then(() => {

        console.log('Tablas sincronizadas')

    });

module.export = {

    Clientes

};
```

---

Debemos crear una carpeta llamada **models**, donde se crearan cada uno de los modelos o tablas a utilizar. Estando en models creamos todos los archivos referente a cada una de las tablas de nuestro proyecto.

---

**Creamos un archivo dentro de models llamado clientes.js**

```
module.exports = (sequelize, type) => {

    return sequelize.define('cliente', {

        nro_documento: {

            type: type.INTEGER,

            primaryKey:true

        },

        nombre: type.STRING,

        direccion: type.STRING,

        telefono: type.STRING,
```

```
  })  
};
```

---

## Mocha

[ <https://mochajs.org/> ]

Es un marco de prueba de JavaScript rico en funciones que se ejecuta en Node.js y en el navegador, lo que hace que las pruebas asincrónicas sean simples y divertidas. Las pruebas de Mocha se ejecutan en serie, lo que permite informes flexibles y precisos, al tiempo que asigna las excepciones no detectadas a los casos de prueba correctos. Alojado en GitHub

```
$ npm install --global mocha  
$ npm install --save-dev mocha
```

A partir de la v9.0.0, Mocha requiere Node.js v12.0.0 o más reciente

Igual de importante que escribir features fantásticos para nuestras aplicaciones, librerías o extensiones es que escribamos buenas pruebas unitarias y de integración. Dado que javascript es un lenguaje que ha tomado mucho auge en los últimos años, es importante que aprendamos a crear pruebas unitarias de una forma correcta y elegante para nuestros proyectos.

En este tutorial pretendo explicarles como utilizar la librería mochaJS para realizar pruebas unitarias sobre nuestro código.

Mocha es un framework de pruebas escrito en javascript que puede correr tanto en node.js como en un navegador, este hace que escribir pruebas para nuestro código de manera asíncrona sea fácil y rápido.

<https://ricardogeek.com/pruebas-unitarias-con-javascript-y-mocha/>

A menudo, cuando desarrollamos una API, nos preguntamos qué podemos utilizar para hacer las pruebas. Este post va a explicar cómo llevar a cabo las pruebas de las principales peticiones HTTP

(GET, POST, DELETE...) sobre una API node utilizando el framework de test MOCHA, la librería de aserciones CHAI y la librería que nos facilita las peticiones HTTP, Chai HTTP.

Aunque ya hablamos en el blog sobre cómo crear pruebas unitarias para nuestro desarrollo en JavaScript con Mocha y Chai, repasamos, de manera breve, qué es Mocha y Chai para ponernos en contexto.

MOCHA es un framework de pruebas para Node JS que puede ejecutarse desde la consola o desde un navegador. Como su propia web indica, permite realizar pruebas asíncronas de manera sencilla y divertida. Al ejecutar los test, permite la presentación de informes flexibles y precisos.

CHAI es una librería de aserciones BDD/TDD para Node JS y navegador, que puede ser armónicamente emparejado con cualquier framework Javascript.

Chai HTTP es una extensión de la librería CHAI, que permite realizar pruebas de integración con llamadas HTTP utilizando las aserciones de CHAI y todos los métodos de HTTP: GET, POST, PUT, DELETE, PATCH...

Para entender mejor cómo funciona Chai HTTP, se ha subido al GitHub de Paradigma un proyecto escrito en Node JS que contiene una API REST y las pruebas implementadas con Mocha y Chai HTTP.

La API generada permite gestionar un listado de países, cada uno de ellos se compone de un ID, un nombre, un año y número de días. En el proyecto dentro de la carpeta *test*, disponemos del

fichero *testChaiHTTP.js* que contiene ejemplos de POST, GET, PUT y DELETE. Veamos cada uno de ellos.

## POST

En el fichero *testChaiHTTP.js* disponemos de dos test que realizan un post a nuestra API.

Primer ejemplo:

Primero requerimos los paquetes necesarios:

```
let chai = require('chai');  
  
let chaiHttp = require('chai-http');  
  
const expect = require('chai').expect;
```

Copy.

Una vez que tenemos los paquetes requeridos, tenemos que decirle a Chai que utilice la librería de Chai HTTP y definimos la url donde vamos a lanzar las llamadas a la API.

```
chai.use(chaiHttp);
```

```
const url= 'http://localhost:3000';
```

Copy.

Para montar el test con Mocha, primero encapsulamos el test dentro de la función *describe*, donde vamos a introducir una descripción del test que se va a realizar.

Dentro de dicha función llamamos a la función *it*, que es donde vamos a explicar lo que queremos que haga el test.

```
describe('Insert a country:',()=>{
```

```
  it('should insert a country', (done) => {
```

```
    chai.request(url)
```

```
      .post('/country')
```

```
.send({id:0, country: "Croacia", year: 2017, days: 10})
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.status(200);
```

```
done();
```

```
});
```

```
});
```

```
});
```

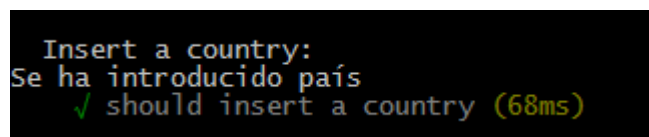


Copy.

Nuestro test realiza un post sobre la API de países que hemos creado. Para insertar un nuevo país a la lista de países, se realiza un `chai.request` a la url que hemos definido, utilizando el método `post` de Chai HTTP, mandando el nuevo país en formato JSON.

Por último chequeamos con el método `end`, en cuya respuesta nos está devolviendo un mensaje en el body y un código 200.

Si ejecutamos el test obtenemos el siguiente resultado:



```
Insert a country:  
Se ha introducido país  
✓ should insert a country (68ms)
```

Segundo ejemplo:

En el segundo ejemplo vamos a realizar una llamada errónea a la API intentando introducir un país que no es un país, por lo que el API contestará con un error 500 en este caso.

```
describe('Insert a country with error: ', () => {
```

```
  it('should receive an error', (done) => {
```

```
    chai.request(url)
```

```
.post('/country')
```

```
.send({id:1, country: "Madrid", year: 2010, days: 10})
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.status(500);
```

```
done();
```

```
});
```

```
});
```

```
});
```

Copy.

Si ejecutamos el test obtenemos el siguiente resultado:

```
Insert a country with error:  
Elemento country enviado no es un país  
✓ should receive an error
```

En la segunda línea podemos ver el mensaje de error que ha encapsulado el API.

GET

Tercer ejemplo:

Para este ejemplo vamos a obtener todos los países que tenemos introducidos en la lista de países.

Para ello vamos a utilizar la función *get*.

```
describe('get all countries: ', () => {
```

```
  it('should get all countries', (done) => {
```

```
    chai.request(url)
```

```
.get('/countries')
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.status(200);
```

```
done();
```

```
});
```

```
});
```

```
});
```

Copy.

Si ejecutamos el test obtenemos el siguiente resultado:

```
get all countries:  
[ { id: 0, country: 'Noruega', year: 2015, days: 17 },  
  { id: 1, country: 'Croacia', year: 2017, days: 10 } ]  
✓ should get all countries
```

La API nos devuelve el listado con todos los países introducidos.

Cuarto ejemplo:

En este ejemplo vamos a utilizar también la función *get*, pero en este caso vamos a obtener un elemento concreto de la lista.

```
describe('get the country with id 1:', () => {
```

```
  it('should get the country with id 1', (done) => {
```

```
    chai.request(url)
```

```
.get('/country/1')
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res.body).to.have.property('id').to.be.equal(1);
```

```
expect(res).to.have.status(200);
```

```
done();
```

```
});
```

```
});
```

```
});
```

Copy.

Si ejecutamos el test obtenemos el siguiente resultado:

```
get the country with id 1:  
{ id: 1, country: 'Croacia', year: 2017, days: 10 }  
✓ should get the country with id 1
```

La API nos devuelve el país con ID 1.

PUT

Quinto ejemplo:

En este ejemplo se va a realizar una llamada a la función PUT para actualizar el valor de los días de un país, pasándole en la URL el ID del país y los días a aumentar.

```
describe('update the days of country with id 1: ', () => {
```

```
  it('should update the number of days', (done) => {
```

```
    chai.request(url)
```

```
.put('/country/1/days/20')
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res.body).to.have.property('days').to.be.equal(20);
```

```
expect(res).to.have.status(200);
```

```
done();
```

```
});
```

```
});
```

```
});
```



Copy.

Si ejecutamos el test obtenemos el siguiente resultado:

```
update the days of country with id 1:  
{ id: 1, country: 'Croacia', year: 2017, days: 20 }  
✓ should update the number of days
```

Los días de Croacia han aumentado de 10 a 20.

DELETE

Sexto ejemplo:

En este ejemplo, se va eliminar un país de la lista. Primero vamos a obtener todos los países para comprobar que hay países en la lista, después vamos a eliminar el país con ID 1 utilizando la función *del*. Por último, vamos a obtener otra vez todos los países para comprobar que el país con ID 1 se ha eliminado.

```
describe('delete the country with id 1:',()=>{
```

```
it('should delete the country with id 1', (done) => {
```

```
  chai.request(url)
```

```
    .get('/countries')
```

```
    .end( function(err,res){
```

```
      console.log(res.body)
```

```
      expect(res.body).to.have.lengthOf(2);
```

```
      expect(res).to.have.status(200);
```

```
    } )
```

```
    .del('/country/1')
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.status(200);
```

```
chai.request(url)
```

```
.get('/countries')
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res.body).to.have.lengthOf(1);
```

```
expect(res.body[0]).to.have.property('id').to.be.equal(0);
```

```
expect(res).to.have.status(200);
```

```
done();
```

```
});
```

```
});
```

```
});
```

```
});
```

```
});
```

Copy.

Si ejecutamos el test obtenemos el siguiente resultado:

```
delete the country with id 1:
[ { id: 0, country: 'Noruega', year: 2015, days: 17 },
  { id: 1, country: 'Croacia', year: 2017, days: 20 } ]
Elemento eliminado
[ { id: 0, country: 'Noruega', year: 2015, days: 17 } ]
✓ should delete the country with id 1
```

Al principio, tenemos los países con ID 0 e id 1 y después de eliminar el país con ID 1, revisamos la lista de países y comprobamos que el país con ID 1 no existe.

FORM

Séptimo Ejemplo

En el séptimo ejemplo vamos a ver cómo se enviaría mediante un POST, datos en forma de formulario. Para ello utilizamos *type*, que va a indicar que vamos a mandar los datos como un formulario.

```
describe('Insert a country with a form: ', () => {
```

```
  it('should receive an error because we send the country in form format', (done) => {
```

```
    chai.request(url)
```

```
      .post('/country')
```

```
.type("form")
```

```
.send({id:0, country: "Croacia", year: 2017, days: 10})
```

```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.status(500);
```

```
done();
```

```
});
```

```
});
```

```
});
```

Copy.

Como resultado del test obtenemos el siguiente resultado:

```
Insert a country with a form:  
Elemento viene como un formulario. Solo se admite JSON  
✓ should receive an error because we send the country in for
```

El resultado que obtenemos es un error ya que no estamos dando los datos en el formato esperado, un JSON. Para comprobar el formato en el que se han mandado los datos, hemos puesto una traza en el servicio que expone la API para que se pueda observar el formato enviado.

```
-----  
id=0&country=Croacia&year=2017&days=10  
-----
```

Datos enviados como formulario.

AGENT and COOKIES

Octavo ejemplo:

En este ejemplo se va a explicar el uso de un agent para realizar una autenticación en el sistema y a su vez, utilizar la cookie que nos proporciona el sistema de autenticación para hacer otra llamada.

Para ello primero declaramos el agent pasándole la url de la API:

```
var agent = chai.request.agent(url)
```

Copy.

Después realizamos una autenticación básica en el sistema y obtenemos la cookie *authToken*. Una vez obtenida la cookie, gracias al agent que hemos creado no tenemos que volver a enviarla, el la almacena y en la siguiente llamada proporciona el token de autenticación que nos proporcionó el sistema:

```
describe('Authenticate a user: ',()=>{
```

```
  it('should receive an OK and a cookie with the authentication token', (done) => {
```

```
    agent
```

```
    .get('/authentication')
```

```
    .auth('user', 'password')
```



```
.end( function(err,res){
```

```
console.log(res.body)
```

```
expect(res).to.have.cookie('authToken');
```

```
expect(res).to.have.status(200);
```

```
return agent.get('/personalData/user')
```

```
.then(function (res) {
```

```
expect(res).to.have.status(200);
```

```
console.log(res.body)
```

```
done();
```

```
});
```

```
done();
```

```
});
```

```
});
```

```
});
```

Copy.

Al ejecutar el test obtenemos el siguiente resultado:

```
Authenticate a user:
El usuario se encuentra en la BBDD
{ id: 0, user: 'user', name: 'Cachimiro', age: '28' }
✓ should receive an OK and a cookie with the authentication
```

Como podemos observar en la ejecución del test obtenemos el OK del servidor que nos indica que el usuario se encuentra en la BBDD y nos proporciona el token de autenticación a través de la cookie. Realizamos la segunda llamada para obtener los datos personales del usuario y obtenemos un JSON con los datos.

El agent almacena la cookie sólo para un test. Para comprobar este comportamiento se ha realizado un test que comprueba que si hacemos la misma llamada para obtener los datos personales del usuario, sin realizar antes la autenticación, nos devuelve un error 500 el servidor.

```
describe('Obtain personal data without authToken: ', () => {  
  
  it('should receive an error because we need authToken', (done) => {  
  
    agent  
  
    .get('/personalData/user')
```

```
.then(function (res) {  
  
    expect(res).to.have.status(500);  
  
    console.log(res.body)  
  
    });  
  
done();  
  
});  
  
});
```

Copy.

El resultado de la ejecución del test es el siguiente:

```
Obtain personal data without authToken:  
✓ should receive an error because we need authToken
```

El servidor retorna un código 500 informando de que se ha producido un error al intentar acceder a los datos sin estar autenticado.

Otras Funcionalidades

Chai HTTP es una herramienta de testeo de APIs bastante robusta, además de los ejemplos básicos que se han realizado, permite encapsular en las peticiones bastantes más objetos como por ejemplo:

- Enviar una cabecera:

```
.set('X-API-Key', 'foobar')
```

Copy.

- Enviar los datos como un formulario:

```
.type('form')
```

```
.send({
```

```
'_method': 'put',
```

```
'password': '123',
```

```
'confirmPassword': '123'
```

```
}
```

Copy.

- Adjuntar parámetros de consulta:

```
.query({name: 'foo', limit: 10})
```

Copy.

- Adjuntar ficheros:

```
.attach('imageField', fs.readFileSync('avatar.png'), 'avatar.png')
```

Copy.

- Chequear si hemos recibido alguna cookie:

```
expect(res).to.have.cookie('sessionid');
```

Copy.

Estos son algunos ejemplos de funcionalidades que Chai HTTP permite. Para más información, puedes consultar su [GitHub](#).

#### Exclusivas de Chai HTTP

Chai HTTP también cuenta con una serie de aserciones exclusivas que se pueden combinar con cualquiera de las que ya incluye Chai. Algunas ya las hemos visto cómo *status* que comprueba el código de estado de la petición o la aserción *cookie* que comprueba si dispone de una cookie. Pero hay muchas más como por ejemplo:

- **Header:** Que comprueba las cabeceras que se han recibido en la respuesta.



```
expect(req).to.have.header('x-api-key');
```

Copy.

- JSON o HTML o TEST permiten comprobar el *content type* de la respuesta recibida.

```
expect(req).to.be.json;
```

```
expect(req).to.be.html;
```

```
expect(req).to.be.text;
```

Copy.

- RedirectTo: permite redireccionar una respuesta a una url destino especificada.

```
expect(res).to.redirectTo('http://example.com');
```

Copy.

Estos son algunos ejemplos, para más información consultar su GitHub.

Ejecución

Para poder lanzar los test o correr el API necesitamos instalar Node Js.

Una vez instalado Node JS y descargado el proyecto, nos situamos en la carpeta del proyecto y ejecutamos:

```
npm install
```

Copy.

Una vez instaladas las dependencias, ejecutamos el servidor que contiene nuestra API escuchando en el puerto 3000, con el comando:

```
node server.js
```

Copy.

Para lanzar los test realizados se tiene que introducir el comando:

```
mocha test/*.js --timeout 15000
```

Copy.

```
Insert a country:
Se ha introducido país
  ✓ should insert a country (68ms)

Insert a country with error:
Elemento country enviado no es un país
  ✓ should receive an error

get all countries:
[ { id: 0, country: 'Noruega', year: 2015, days: 17 },
  { id: 1, country: 'Croacia', year: 2017, days: 10 } ]
  ✓ should get all countries

get the country with id 1:
{ id: 1, country: 'Croacia', year: 2017, days: 10 }
  ✓ should get the country with id 1

update the days of country with id 1:
{ id: 1, country: 'Croacia', year: 2017, days: 20 }
  ✓ should update the number of days

delete the country with id 1:
[ { id: 0, country: 'Noruega', year: 2015, days: 17 },
  { id: 1, country: 'Croacia', year: 2017, days: 20 } ]
Elemento eliminado
[ { id: 0, country: 'Noruega', year: 2015, days: 17 } ]
  ✓ should delete the country with id 1

6 passing (124ms)
```

Conclusión

Finalmente la elección de la librería de pruebas se elige en función del lenguaje de programación con el que se desarrolla la API, para que todo esté unificado bajo un mismo lenguaje de programación.

Por lo que si la API está escrita en NodeJS, personalmente recomiendo utilizar Chai HTTP con Mocha y Chai porque son herramientas compatibles, fáciles de utilizar y rápidas.

Se pueden encontrar bastantes casos de ejemplo y dudas resueltas en Internet lo que hace que sea una herramienta utilizada por la comunidad.

Pero no sólo de Chai HTTP vive el hombre, hay muchas alternativas para realizar pruebas sobre APIs por lo que invito a utilizar y experimentar todas y adaptar la que más guste al equipo.