Table of Contents

Introducción

Objetivos del curso

- Conocer la forma de desarrollar con node.js
  - Configuración del editor de código
  - Uso de librerías
  - Publicación de módulos mediante npm
- Acceso a bases de datos MongoDB
  - Concepto de ODM
  - Utilizar mongodb para acceder a datos
  - Utilizar Mongoose para acceder a datos
  - Ejemplos de uso. Casos prácticos con las ventajas de Mongoose
- Creación de una API mediante node.js

Documentación

- **Generada con Gitbook**
- Para generar las slides, pdf, epub o mobi
  - Es necesario tener instalado calibre

```
git clone https://github.com/juanda99/node-mongodb
cd node-mongodb
node_modules/.bin/gitbook install
npm install -g gitbook-cli
npm install
npm run slides
npm run pdf
npm run epub
npm run mobi
```

- Libro online con generación de pdf, epub y mobi

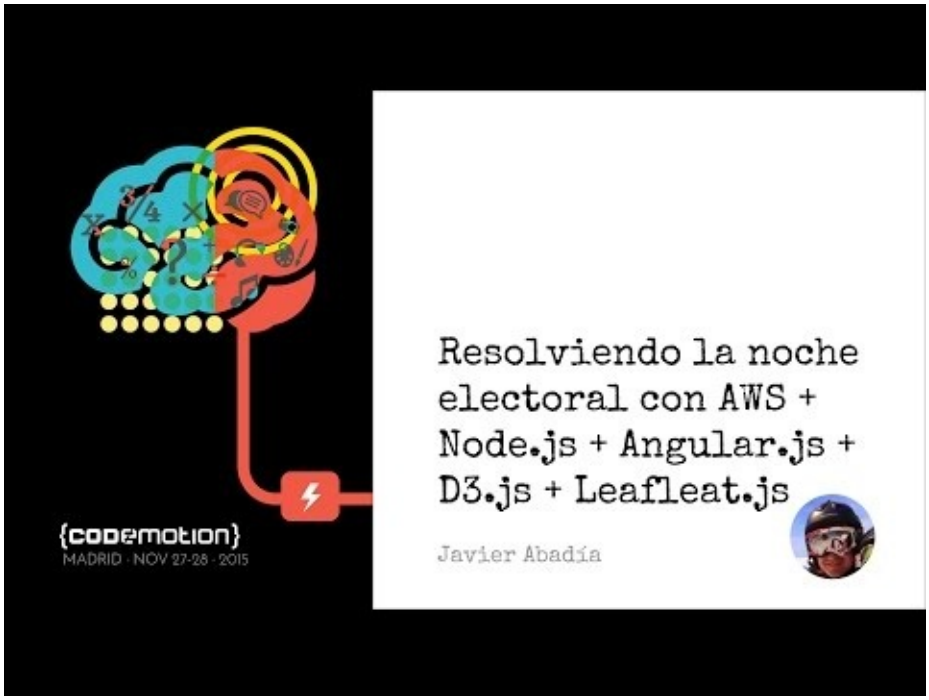
JavaScript en servidor

Qué es nodejs

- NodeJS es un intérprete de JavaScript que se ejecuta en servidor.
- Está basado en el motor de JavaScript que utiliza Google Chrome (V8), escrito en C++

Características principales

- El tener el mismo lenguaje en cliente y servidor
  - Permite a cualquier persona desarrollar en backend o en frontend
  - Permite reusar código o incluso mover código de cliente a servidor o al revés
- Está orientado a eventos y utiliza un modelo asíncrono (propio de JavaScript).
- Al contrario que en el navegador, encontramos muchas llamadas asíncronas:
  - Llamadas a APIs
  - Lectura y escritura de ficheros
  - Ejecución de cálculos en el servidor
  - ....
- Llamadas síncronas en servidor serían fatales:
  - ¡Bloquearíamos las conexiones al servidor hasta que acabase la instrucción bloqueante!
  - Al ser asíncrono podremos tener muchas sesiones concurrentes
- Es monohilo
  - Utiliza un solo procesador
  - Si queremos usar toda la potencia de la CPU, tendremos que levantar varias instancias de node y utilizar un balanceador de carga (por ejemplo con pm2)



Video link

Ver la parte interesante del video: mínuto 14:47

Desventajas

- Trabajar con código asíncrono hace que a veces el código no sea excesivamente legible
- Imagina que guardamos un registro de los accesos de los usuarios a nuestra app:

```
trackUser =  function(userId) {
  users.findOne({userId: userId}, function(err, user) {
    var logIn = {userName: user.name, when: new Date};
    logIns.insert(logIn, function(err, done) {
      console.log('wrote log-in!');
      });
  });
```

- Tenemos 3 funciones anidadas en una simple operación.
- Esto es lo que se conoce como callback hell

Evitar el callback en el navegador

- Mediante el uso de promesas
- Se trata de escribir código asíncrono con un estilo síncrono.
- Opciones más actuales:
  - Generators / Yields (ES6)
  - Async / Await (ES7)
  - El soporte de ES6 en node es limitado (--harmony) y también en el navegador => TRANSPILERS (babel)
- Ver comparativa de métodos asíncronos

Hola Mundo en node

- Editamos un fichero en JavaScript, *holaMundo.js*:

```
console.log ("Hola Mundo");
```

- Lo ejecutamos mediante *node holaMundo.js*
- Si escribimos *node* sin más, podemos acceder a la consola de node, un intérprete de JavaScript, igual que el que tenemos en el navegador

npm

- Es el gestor de paquetes de node
- Propongo hacer dos prácticas para coger la dinámica del uso de npm y sus librerías y de trabajar con node:
  - Crear una librería en node.js
  - Crear una api rest mediante node.js

Crear una librería en node.JS

Librerías en node

- Suelen ser pequeñas
- Es un buen ejemplo de ciclo de desarrollo en node.js
- Ayuda a tener claro el concepto de paquetes de node

Microlibrerías

- Ventajas
  - Poco código, se entiende y modifica con facilidad
  - Reusable
  - Fácil hacer tests
- Desventajas
  - Tienes que gestionar muchas dependencias
  - Control de versiones de todas ellas

Funcionalidad librería

- Registrar un nuevo usuario
- Validar un usuario en base a:
  - email y password
  - auth token
- Eliminar usuario
- Activar usuario

Control de versiones

- Utilizaremos git como control de versiones
- Utilizaremos github como servidor git en la nube para almacenar nuestro repositorio:
  - Haz login con tu usuario (o crea un usuario nuevo)
  - Crea un nuevo repositorio en GitHub (lo llamaré *user-auth*)
  - Sigue las indicaciones de GitHub para crear el repositorio en local y asociarlo al repositorio remoto (GitHub)

Creamos el proyecto

- Dentro del directorio user-auth:

```
npm init
```

- El *entry-point* lo pondremos en *src/index.js*, así separaremos nuesto código fuente de los tests.
- El resto de parámetros con sus valores por defecto
- ¡Ya tenemos nuestro **package.json** creado!

Crear modelo usuario

- Para operar con usuarios (documentos) en nuestro código necesitamos objetos JS
- Instanciamos un modelo de usuario de MongoDB que será sobre el que operaremos (lo más cómodo)

```
var User = mongoose.model('User', userSchema);
```

- Los usuarios se guardarán en la colección Users
- Debemos definir el userSchema

Creación del esquema de usuario

- Definimos los campos que necesitemos
- Definimos las validaciones necesarias

```
var UserSchema = new Schema({
  email: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  username: {
    type: String,
    unique: true,
    required: true,
    trim: true
  },
  password: {
    type: String,
    required: true,
  }
});
```

Encriptación de la contraseña

- La contraseña debe ir encriptada
- Usaremos una librería que se encargue de ello: bcrypt
- Definimos un middleware que actúe antes del evento save

var mongoose = require('mongoose'); var Schema = mongoose.Schema;

var mongoose = require('mongoose'); var UserSchema = new Schema({ email: { type: String, unique: true, required: true, trim: true }, username: { type: String, unique: true, required: true, trim: true }, password: { type: String, required: true, } }); var User = mongoose.model('User', UserSchema); module.exports = User;

Ahora queremos obtener una cerveza al azar:

- Instalamos el paquete uniqueRandomArray

  ```
  npm i -S unique-random-array
  ```

- Configuramos nuestro fuente:

```
cervezas = require('./cervezas.json')
var uniqueRandomArray = require ('unique-random-array')
module.exports = {
  todas: cervezas,
  alazar: uniqueRandomArray(cervezas)
}
```

- Comprobamos que funcione. Ojo, ¡alazar es una función!

Subimos la librería a github

- Necesitamos crear un **.gitignore** para la carpeta no sincronizar **node_modules**
- Los comandos que habrá que hacer luego son:

```
git status
git add -A
git status
git commit -m "versión inicial"
```

- Ojo que haya cogido los cambios del .gitignore para hacer el push

```
git push
```

- Comprobamos ahora en github que esté todo correcto.

Publicamos en npm

```
npm publish
```

- Podemos comprobar la información que tiene npm de cualquier paquete mediante

```
npm info <nombre paquete>
```

Probamos nuestra librería

- Creamos un nuevo proyecto e instalamos nuestra librería
- Creamos un index para utilizarla:

```
var cervezas = require('cervezas')
console.log(cervezas.alazar())
console.log(cervezas.todas)
```

- Ejecutamos nuestro fichero:

```
node index.js
```

Versiones en GitHub

- Nuestro paquete tiene la versión 1.0.0 en npm
- Nuestro paquete no tiene versión en GitHub, lo haremos mediante el uso de etiquetas:

```
git tag v1.0.0
```

```
git push --tags
```

- Comprobamos ahora que aparece en la opción Releases y que la podemos modificar.
- También aparece en el botón de seleccionar branch, pulsando luego en la pestaña de tags.

Modificar librería

- Queremos mostrar las cervezas ordenadas por nombre
- Utilizaremos la **librería lodash** (navaja suiza del js) para ello:

```
var cervezas = require('./cervezas.json');
var uniqueRandomArray = require('unique-random-array');
var _ = require('lodash');
module.exports = {
  todas: _.sortBy(cervezas, ['nombre']),
  alazar: uniqueRandomArray(cervezas)
}
```

- Ahora tendremos que cambiar la versión a 1.1.0 (semver) en el package.json y publicar el paquete de nuevo
- También añadiremos la tag en GitHub ¿Lo vamos pillando?

Versiones beta

- Vamos a añadir una cerveza nueva, pero todavía no se está vendiendo.
- Aumentamos nuestra versión a 1.2.0-beta.0 (nueva funcionalidad, pero en beta)
- Al subirlo a npm:

```
npm publish --tag beta
```

- Con npm info podremos ver un listado de nuestras versiones (¡mirá las dist-tags)
- Para instalar la versión beta:

```
npm install <nombre paquete>@beta
```

Tests

- Utilizaremos **Mocha** y **Chai**
- Las instalaremos como dependencias de desarrollo:

```
npm i -D mocha chai
```

- Añadimos el comando para test en el package.json (-w para que observe):

```
 "test": "mocha src/index.test.js -w"
```

- Creamos un fichero src/index.test.js con las pruebas

```
var expect = require('chai').expect;
describe('cervezas', function () {
  it('should work!', function (done) {
      expect(true).to.be.true;
      done();
  });
});
```

- Utiliza los paquetes **Mocha Snippets** y **Chai Completions** de Sublime Text para completar el código
- Ahora prepararemos una estructura de tests algo más elaborada:

```
var expect = require('chai').expect;
var cervezas = require('./index');

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            // se comprueba que cumpla la condición de ser array de objetos
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            // se comprueba que incluya la cerveza Ambar
            done();
        });
    });
    describe('alazar', function () {
        it('Debería mostrar una cerveza de la lista', function (done) {
            //
            done();
        });
    });
});
```

- Por último realizamos los tests:

```
var expect = require('chai').expect;
var cervezas = require('./index');
var _ = require('lodash')

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            expect(cervezas.todas).to.satisfy(isArrayOfObjects);
            function isArrayOfObjects(array){
                return array.every(function(item){
                    return typeof item === 'object';
                });
            }
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            expect(cervezas.todas).to.satisfy(contieneAmbar);
            function contieneAmbar (array){
                return _.some(array, { 'nombre': 'ÁMBAR ESPECIAL' });
            }
            done();
        });

    });
    describe('alazar', function () {
        it('Debería mostrar un elemento de la lista de cervezas', function (done) {
            var cerveza = cervezas.alazar();
            expect(cervezas.todas).to.include(cerveza);
            done();
        });
    });
});
```

Automatizar tareas

- Cada vez que desarrollamos una versión de nuestra libería:

    - Ejecutar los tests
    - Hay que realizar un commit
    - Hay que realizar un tag del commig
    - Push a GitHub
    - Publicar en npm
    - ...
- Vamos a intentar automatizar todo:
    - **Semantic Release** para la gestión de versiones
    - **Travis** como CI (continuous integration)

Instalación Semantic Release

- Paso previo (en Ubuntu 14.04, si no fallaba la instalación):

```
sudo apt-get install libgnome-keyring-dev
```

- Instalación y configuración:

```
sudo npm i -g semantic-release-cli
semantic-release-cli setup
```

- **.travis.yml**: contiene la configuración de Travis
- Cambios en package.json:
    - Incluye un nuevo script (*semantic-release*)
    - Quita la versión
    - Añade la dependencia de desarrollo de Semantic Release

Versiones del software

- Utilizamos semantic versioning
- Semantic Release se ejecuta a través de Travis CI
- Travis CI se ejecuta al hacer un push (hay que configurarlo desde la web)
- Los commit tienen que seguir las reglas del equipo de Angular

Uso de commitizen

- **commitizen** que nos ayudará en la generación de los mensajes de los commit.
- La instalación, siguiendo su documentación:

```
sudo npm install commitizen -g
commitizen init cz-conventional-changelog --save-dev --save-exact
```

- Habrá que ejecutar **git cz** en vez de **git commit** para que los commits los gestione commitizen

Cambio de versión

- Vamos a comprobar nuestro entorno añadiendo una funcionalidad
- Si pedimos cervezas.alazar() queremos poder recibir más de una

- Los tests:

```
it('Debería mostrar varias cervezas de la lista', function (done) {
    var misCervezas = cervezas.alazar(3);
    expect(misCervezas).to.have.length(3);
    misCervezas.forEach(function(cerveza){
        expect(cervezas.todas).to.include(cerveza);
    });
    done();
});
```

- Añadimos la funcionalidad en el *src/index.js*: ``` var cervezas = require('./cervezas.json'); var uniqueRandomArray = require('unique-random-array'); var = *require('lodash'); var getCerveza = uniqueRandomArray(cervezas) module.exports = { todas:* .sortBy(cervezas, ['nombre']), alazar: alazar }

function alazar(unidades) { if (unidades===undefined){ return getCerveza(); } else { var misCervezas = []; for (var i = 0; i<unidades; i++) { misCervezas.push(getCerveza()); } return misCervezas; } }

```
- Hagamos ahora el git cz & git push y veamos como funciona todo
- Podríamos añadir un issue y hacer el fix en este commit escribiendo closes #issue en el footer del commit mes
sage.

## Git Hooks
- Son una manera de ejecutar scripts antes de que ocurra alguna acción
- Sería ideal pasar los tests antes de que se hiciera el commit
- Los Git Hooks son locales:
    - Si alguien hace un clone del repositorio, no tiene los GitHooks
    - Instalaremos un paquete de npm para hacer git hooks de forma universal
```

npm i -D ghooks

```
- Lo configuraremos en el package.json en base a la [documentación del paquete](https://www.npmjs.com/package/g
hooks):
```

"config": { "ghooks": { "pre-commit": "npm test" } }

```
## Coverage
- Nos interesa que todo nuestro código se pruebe mediante tests.
- Necesitamos una herramienta que compruebe el código mientras se realizan los tests:
```

npm i -D instanbul

```
- Modificaremos el script de tests en el package.json:
```

istanbul cover -x *.test.js _mocha -- -R spec src/index.test.js

```
- Instanbul analizará la cobertura de todos los ficheros excepto los de test ejecutando a su vez _mocha (un wra
pper de mocha proporcionado por ellos) con los tests.
- Si ejecutamos ahora *npm test* nos ofrecerá un resumen de la cobertura de nuestros tests.
- Por último nos crea una carpeta en el proyecto *coverage* donde podemos ver los datos, por ejemplo desde un n
avegador (fichero index.html)
- ¡Ojo, recordar poner la carpeta coverage en el .gitignore!

## Check coverage
- Podemos también evitar los commits si no hay un porcentaje de tests óptimo:
```

"pre-commit": "npm test && npm run check-coverage"

```
- Creamos el script check-coverage dentro del package.json:
```

"check-coverage": "istanbul check-coverage --statements 100 --branches 100 --functions 100 -lines 100"

```
- Podemos comprobar su ejecución desde el terminal mediante *npm run check-coverage* y añadir una función nueva
 sin tests, para comprobar que el check-coverage no termina con éxito.
- Lo podemos añadir también en Travis, de modo que no se haga una nueva release si no hay ciertos estándares (e
l test si lo hace por defecto):
```

script:

- npm run test
- npm run check-coverage ```

Gráficas

- Utilizaremos la herramienta codecov.io:

  ```
  npm i -D codecov.io
  ```

- Crearemos un script que recoge los datos de istanbul:

  ```
  "report-coverage": "cat ./coverage/lcov.info | codecov"
  ```

- Lo añadimos en travis de modo que genere un reporte:

  ```
  after success:
  - npm run report-coverage
  - npm run semantic-release
  ```

- Integrado con github (chrome extension)
- Por último podemos añadir etiquetas de muchos servicios: npm, codecov, travis... una fuente habitual es http://www.shields.io

Crear una librería en node.JS

Librerías en node

- Suelen ser pequeñas
- Es un buen ejemplo de ciclo de desarrollo en node.js
- Ayuda a tener claro el concepto de paquetes de node

Microlibrerías

- Ventajas
  - Poco código, se entiende y modifica con facilidad
  - Reusable
  - Fácil hacer tests

- Desventajas
  - Tienes que gestionar muchas dependencias
  - Control de versiones de todas ellas

Funcionalidad librería

- Obtiene una marca de cerveza y sus características
- Obtiene una o varias marcas de cerveza al azar.

Control de versiones

- Utilizaremos git como control de versiones
- Utilizaremos github como servidor git en la nube para almacenar nuestro repositorio:
  - Haz login con tu usuario (o crea un usuario nuevo)
  - Crea un nuevo repositorio en GitHub (lo llamaré *cervezas*)
  - Sigue las indicaciones de GitHub para crear el repositorio en local y asociarlo al repositorio remoto (GitHub)

Instalación de node

- Lo más sencillo es instalar mediante el gestor de paquetes

```
curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
sudo apt-get install -y nodejs
```

- Comprobamos que esté correctamente instalado

```
node -v
npm -v
```

npm

- Es el gestor de paquetes de node
- **Debemos crear un usuario** en https://www.npmjs.com/
- Podemos buscar los paquetes que nos interese instalar
- Podemos publicar nuestra librería :-)

Configuración de npm

- Cuando creemos un nuevo proyecto nos interesa que genere automaticamente datos como nuestro nombre o email
- Ver documentación para su configuación o mediante consola (`npm --help`) :
- Mediante `npm config --help` vemos los comandos de configuración
- Mediante `npm config ls -l` vemos los parámetros de configuración

```
npm set init-author-name pepe
npm set init-author-email pepe@pepe.com
npm set init-author-url http://pepe.com
npm set init-license MIT
npm adduser
```

- Los cambios se guardan en el fichero $HOME/.npmrc
- `npm adduser` genera un authtoken = login automático al publicar en el registro de npm

Versiones en node

- Se utiliza Semantic Versioning

```
npm set save-exact true
```

- Las versiones tienen el formato MAJOR.MINOR.PATCH
- Cambios de versión:
  - MAJOR: Cambios en compatibilidad de API,
  - MINOR: Se añade funcionalidad. Se mantiene la compatibilidad.
  - PATCH: Se solucionan bug. Se mantiene compatibilidad.
- ¡Puede obligarnos a cambiar el MAJOR muy a menudo!

Creamos el proyecto

- Dentro del directorio cervezas:

```
npm init
```

- El *entry-point* lo pondremos en *src/index.js*, así separaremos nuestro código fuente de los tests.
- El resto de parámetros con sus valores por defecto
- ¡Ya tenemos nuestro **package.json** creado!

Listar todas las cervezas:

- Editamos nuestro fichero *src/index.js*

```
var cervezas = require('./cervezas.json')
module.exports = {
  todas: cervezas
}
```

- Abrimos una consola y comprobamos que funcione nuestra librería:

```
node
> var cervezas = require('./index.js')
undefined
> cervezas.todas
```

Ahora queremos obtener una cerveza al azar:

- Instalamos el paquete uniqueRandomArray

```
npm i -S unique-random-array
```

- Configuramos nuestro fuente:

```
cervezas = require('./cervezas.json')
var uniqueRandomArray = require ('unique-random-array')
module.exports = {
  todas: cervezas,
  alazar: uniqueRandomArray(cervezas)
}
```

- Comprobamos que funcione. Ojo, ¡alazar es una función!

Subimos la librería a github

- Necesitamos crear un **.gitignore** para la carpeta no sincronizar **node_modules**
- Los comandos que habrá que hacer luego son:

```
git status
git add -A
git status
git commit -m "versión inicial"
```

- Ojo que haya cogido los cambios del .gitignore para hacer el push

```
git push
```

- Comprobamos ahora en github que esté todo correcto.

Publicamos en npm

```
npm publish
```

- Podemos comprobar la información que tiene npm de cualquier paquete mediante

```
npm info <nombre paquete>
```

Probamos nuestra librería

- Creamos un nuevo proyecto e instalamos nuestra librería
- Creamos un index para utilizarla:

```
var cervezas = require('cervezas')
console.log(cervezas.alazar())
console.log(cervezas.todas)
```

- Ejecutamos nuestro fichero:

```
node index.js
```

Versiones en GitHub

- Nuestro paquete tiene la versión 1.0.0 en npm
- Nuestro paquete no tiene versión en GitHub, lo haremos mediante el uso de etiquetas:

```
git tag v1.0.0
```

```
git push --tags
```

- Comprobamos ahora que aparece en la opción Releases y que la podemos modificar.
- También aparece en el botón de seleccionar branch, pulsando luego en la pestaña de tags.

Modificar librería

- Queremos mostrar las cervezas ordenadas por nombre
- Utilizaremos la **librería lodash** (navaja suiza del js) para ello:

```
var cervezas = require('./cervezas.json');
var uniqueRandomArray = require('unique-random-array');
var _ = require('lodash');
module.exports = {
  todas: _.sortBy(cervezas, ['nombre']),
  alazar: uniqueRandomArray(cervezas)
}
```

- Ahora tendremos que cambiar la versión a 1.1.0 (semver) en el package.json y publicar el paquete de nuevo
- También añadiremos la tag en GitHub ¿Lo vamos pillando?

Versiones beta

- Vamos a añadir una cerveza nueva, pero todavía no se está vendiendo.
- Aumentamos nuestra versión a 1.2.0-beta.0 (nueva funcionalidad, pero en beta)
- Al subirlo a npm:

```
npm publish --tag beta
```

- Con npm info podremos ver un listado de nuestras versiones (¡mirá las dist-tags)
- Para instalar la versión beta:

```
npm install <nombre paquete>@beta
```

Tests

- Utilizaremos **Mocha** y **Chai**
- Las instalaremos como dependencias de desarrollo:

```
npm i -D mocha chai
```

- Añadimos el comando para test en el package.json (-w para que observe):

```
 "test": "mocha src/index.test.js -w"
```

- Creamos un fichero src/index.test.js con las pruebas

```
var expect = require('chai').expect;
describe('cervezas', function () {
  it('should work!', function (done) {
      expect(true).to.be.true;
      done();
  });
});
```

- Utiliza los paquetes **Mocha Snippets** y **Chai Completions** de Sublime Text para completar el código
- Ahora prepararemos una estructura de tests algo más elaborada:

```
var expect = require('chai').expect;
var cervezas = require('./index');

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            // se comprueba que cumpla la condición de ser array de objetos
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            // se comprueba que incluya la cerveza Ambar
            done();
        });
    });
    describe('alazar', function () {
        it('Debería mostrar una cerveza de la lista', function (done) {
            //
            done();
        });
    });
});
```

- Por último realizamos los tests:

```
var expect = require('chai').expect;
var cervezas = require('./index');
var _ = require('lodash')

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            expect(cervezas.todas).to.satisfy(isArrayOfObjects);
            function isArrayOfObjects(array){
                return array.every(function(item){
                    return typeof item === 'object';
                });
            }
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            expect(cervezas.todas).to.satisfy(contieneAmbar);
            function contieneAmbar (array){
                return _.some(array, { 'nombre': 'ÁMBAR ESPECIAL' });
            }
            done();
        });

    });
    describe('alazar', function () {
        it('Debería mostrar un elemento de la lista de cervezas', function (done) {
            var cerveza = cervezas.alazar();
            expect(cervezas.todas).to.include(cerveza);
            done();
        });
    });
});
```

Automatizar tareas

- Cada vez que desarrollamos una versión de nuestra libería:

    - Ejecutar los tests
    - Hay que realizar un commit
    - Hay que realizar un tag del commig
    - Push a GitHub
    - Publicar en npm
    - ...
- Vamos a intentar automatizar todo:
    - **Semantic Release** para la gestión de versiones
    - **Travis** como CI (continuous integration)

Instalación Semantic Release

- Paso previo (en Ubuntu 14.04, si no fallaba la instalación):

```
sudo apt-get install libgnome-keyring-dev
```

- Instalación y configuración:

```
sudo npm i -g semantic-release-cli
semantic-release-cli setup
```

- **.travis.yml**: contiene la configuración de Travis
- Cambios en package.json:
    - Incluye un nuevo script (*semantic-release*)
    - Quita la versión
    - Añade la dependencia de desarrollo de Semantic Release

Versiones del software

- Utilizamos semantic versioning
- Semantic Release se ejecuta a través de Travis CI
- Travis CI se ejecuta al hacer un push (hay que configurarlo desde la web)
- Los commit tienen que seguir las reglas del equipo de Angular

Uso de commitizen

- **commitizen** que nos ayudará en la generación de los mensajes de los commit.
- La instalación, siguiendo su documentación:

```
sudo npm install commitizen -g
commitizen init cz-conventional-changelog --save-dev --save-exact
```

- Habrá que ejecutar **git cz** en vez de **git commit** para que los commits los gestione commitizen

Cambio de versión

- Vamos a comprobar nuestro entorno añadiendo una funcionalidad
- Si pedimos cervezas.alazar() queremos poder recibir más de una

- Los tests:

```
it('Debería mostrar varias cervezas de la lista', function (done) {
    var misCervezas = cervezas.alazar(3);
    expect(misCervezas).to.have.length(3);
    misCervezas.forEach(function(cerveza){
        expect(cervezas.todas).to.include(cerveza);
    });
    done();
});
```

- Añadimos la funcionalidad en el *src/index.js*: ``` var cervezas = require('./cervezas.json'); var uniqueRandomArray = require('unique-random-array'); var = *require('lodash'); var getCerveza = uniqueRandomArray(cervezas) module.exports = { todas:* .sortBy(cervezas, ['nombre']), alazar: alazar }

function alazar(unidades) { if (unidades===undefined){ return getCerveza(); } else { var misCervezas = []; for (var i = 0; i<unidades; i++) { misCervezas.push(getCerveza()); } return misCervezas; } }

```
- Hagamos ahora el git cz & git push y veamos como funciona todo
- Podríamos añadir un issue y hacer el fix en este commit escribiendo closes #issue en el footer del commit mes
sage.

## Git Hooks
- Son una manera de ejecutar scripts antes de que ocurra alguna acción
- Sería ideal pasar los tests antes de que se hiciera el commit
- Los Git Hooks son locales:
    - Si alguien hace un clone del repositorio, no tiene los GitHooks
    - Instalaremos un paquete de npm para hacer git hooks de forma universal
```

npm i -D ghooks

```
- Lo configuraremos en el package.json en base a la [documentación del paquete](https://www.npmjs.com/package/g
hooks):
```

"config": { "ghooks": { "pre-commit": "npm test" } }

```
## Coverage
- Nos interesa que todo nuestro código se pruebe mediante tests.
- Necesitamos una herramienta que compruebe el código mientras se realizan los tests:
```

npm i -D instanbul

```
- Modificaremos el script de tests en el package.json:
```

istanbul cover -x *.test.js _mocha -- -R spec src/index.test.js

```
- Instanbul analizará la cobertura de todos los ficheros excepto los de test ejecutando a su vez _mocha (un wra
pper de mocha proporcionado por ellos) con los tests.
- Si ejecutamos ahora *npm test* nos ofrecerá un resumen de la cobertura de nuestros tests.
- Por último nos crea una carpeta en el proyecto *coverage* donde podemos ver los datos, por ejemplo desde un n
avegador (fichero index.html)
- ¡Ojo, recordar poner la carpeta coverage en el .gitignore!

## Check coverage
- Podemos también evitar los commits si no hay un porcentaje de tests óptimo:
```

"pre-commit": "npm test && npm run check-coverage"

```
- Creamos el script check-coverage dentro del package.json:
```

"check-coverage": "istanbul check-coverage --statements 100 --branches 100 --functions 100 -lines 100"

```
- Podemos comprobar su ejecución desde el terminal mediante *npm run check-coverage* y añadir una función nueva
 sin tests, para comprobar que el check-coverage no termina con éxito.
- Lo podemos añadir también en Travis, de modo que no se haga una nueva release si no hay ciertos estándares (e
l test si lo hace por defecto):
```

script:

- npm run test
- npm run check-coverage ```

Gráficas

- Utilizaremos la herramienta codecov.io:

  ```
  npm i -D codecov.io
  ```

- Crearemos un script que recoge los datos de istanbul:

  ```
  "report-coverage": "cat ./coverage/lcov.info | codecov"
  ```

- Lo añadimos en travis de modo que genere un reporte:

  ```
  after success:
  - npm run report-coverage
  - npm run semantic-release
  ```

- Integrado con github (chrome extension)
- Por último podemos añadir etiquetas de muchos servicios: npm, codecov, travis... una fuente habitual es http://www.shields.io

Crear una librería en node.JS

Librerías en node

- Suelen ser pequeñas
- Es un buen ejemplo de ciclo de desarrollo en node.js
- Ayuda a tener claro el concepto de paquetes de node

Microlibrerías

- Ventajas
  - Poco código, se entiende y modifica con facilidad
  - Reusable
  - Fácil hacer tests

- Desventajas
  - Tienes que gestionar muchas dependencias
  - Control de versiones de todas ellas

Funcionalidad librería

- Obtiene una marca de cerveza y sus características
- Obtiene una o varias marcas de cerveza al azar.

Control de versiones

- Utilizaremos git como control de versiones
- Utilizaremos github como servidor git en la nube para almacenar nuestro repositorio:
  - Haz login con tu usuario (o crea un usuario nuevo)
  - Crea un nuevo repositorio en GitHub (lo llamaré *cervezas*)
  - Sigue las indicaciones de GitHub para crear el repositorio en local y asociarlo al repositorio remoto (GitHub)

Instalación de node

- Lo más sencillo es instalar mediante el gestor de paquetes

```
curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
sudo apt-get install -y nodejs
```

- Comprobamos que esté correctamente instalado

```
node -v
npm -v
```

npm

- Es el gestor de paquetes de node
- **Debemos crear un usuario** en https://www.npmjs.com/
- Podemos buscar los paquetes que nos interese instalar
- Podemos publicar nuestra librería :-)

Configuración de npm

- Cuando creemos un nuevo proyecto nos interesa que genere automaticamente datos como nuestro nombre o email
- Ver documentación para su configuación o mediante consola (`npm --help`) :
- Mediante `npm config --help` vemos los comandos de configuración
- Mediante `npm config ls -l` vemos los parámetros de configuración

```
npm set init-author-name pepe
npm set init-author-email pepe@pepe.com
npm set init-author-url http://pepe.com
npm set init-license MIT
npm adduser
```

- Los cambios se guardan en el fichero $HOME/.npmrc
- `npm adduser` genera un authtoken = login automático al publicar en el registro de npm

Versiones en node

- Se utiliza Semantic Versioning

```
npm set save-exact true
```

- Las versiones tienen el formato MAJOR.MINOR.PATCH
- Cambios de versión:
  - MAJOR: Cambios en compatibilidad de API,
  - MINOR: Se añade funcionalidad. Se mantiene la compatibilidad.
  - PATCH: Se solucionan bug. Se mantiene compatibilidad.
- ¡Puede obligarnos a cambiar el MAJOR muy a menudo!

Creamos el proyecto

- Dentro del directorio cervezas:

```
npm init
```

- El *entry-point* lo pondremos en *src/index.js*, así separaremos nuestro código fuente de los tests.
- El resto de parámetros con sus valores por defecto
- ¡Ya tenemos nuestro **package.json** creado!

Listar todas las cervezas:

- Editamos nuestro fichero *src/index.js*

```
var cervezas = require('./cervezas.json')
module.exports = {
  todas: cervezas
}
```

- Abrimos una consola y comprobamos que funcione nuestra librería:

```
node
> var cervezas = require('./index.js')
undefined
> cervezas.todas
```

Ahora queremos obtener una cerveza al azar:

- Instalamos el paquete uniqueRandomArray

```
npm i -S unique-random-array
```

- Configuramos nuestro fuente:

```
cervezas = require('./cervezas.json')
var uniqueRandomArray = require ('unique-random-array')
module.exports = {
  todas: cervezas,
  alazar: uniqueRandomArray(cervezas)
}
```

- Comprobamos que funcione. Ojo, ¡alazar es una función!

Subimos la librería a github

- Necesitamos crear un **.gitignore** para la carpeta no sincronizar **node_modules**
- Los comandos que habrá que hacer luego son:

```
git status
git add -A
git status
git commit -m "versión inicial"
```

- Ojo que haya cogido los cambios del .gitignore para hacer el push

```
git push
```

- Comprobamos ahora en github que esté todo correcto.

Publicamos en npm

```
npm publish
```

- Podemos comprobar la información que tiene npm de cualquier paquete mediante

```
npm info <nombre paquete>
```

Probamos nuestra librería

- Creamos un nuevo proyecto e instalamos nuestra librería
- Creamos un index para utilizarla:

```
var cervezas = require('cervezas')
console.log(cervezas.alazar())
console.log(cervezas.todas)
```

- Ejecutamos nuestro fichero:

```
node index.js
```

Versiones en GitHub

- Nuestro paquete tiene la versión 1.0.0 en npm
- Nuestro paquete no tiene versión en GitHub, lo haremos mediante el uso de etiquetas:

```
git tag v1.0.0
```

```
git push --tags
```

- Comprobamos ahora que aparece en la opción Releases y que la podemos modificar.
- También aparece en el botón de seleccionar branch, pulsando luego en la pestaña de tags.

Modificar librería

- Queremos mostrar las cervezas ordenadas por nombre
- Utilizaremos la **librería lodash** (navaja suiza del js) para ello:

```
var cervezas = require('./cervezas.json');
var uniqueRandomArray = require('unique-random-array');
var _ = require('lodash');
module.exports = {
  todas: _.sortBy(cervezas, ['nombre']),
  alazar: uniqueRandomArray(cervezas)
}
```

- Ahora tendremos que cambiar la versión a 1.1.0 (semver) en el package.json y publicar el paquete de nuevo
- También añadiremos la tag en GitHub ¿Lo vamos pillando?

Versiones beta

- Vamos a añadir una cerveza nueva, pero todavía no se está vendiendo.
- Aumentamos nuestra versión a 1.2.0-beta.0 (nueva funcionalidad, pero en beta)
- Al subirlo a npm:

```
npm publish --tag beta
```

- Con npm info podremos ver un listado de nuestras versiones (¡mirá las dist-tags)
- Para instalar la versión beta:

```
npm install <nombre paquete>@beta
```

Tests

- Utilizaremos **Mocha** y **Chai**
- Las instalaremos como dependencias de desarrollo:

```
npm i -D mocha chai
```

- Añadimos el comando para test en el package.json (-w para que observe):

```
 "test": "mocha src/index.test.js -w"
```

- Creamos un fichero src/index.test.js con las pruebas

```
var expect = require('chai').expect;
describe('cervezas', function () {
  it('should work!', function (done) {
      expect(true).to.be.true;
      done();
  });
});
```

- Utiliza los paquetes **Mocha Snippets** y **Chai Completions** de Sublime Text para completar el código
- Ahora prepararemos una estructura de tests algo más elaborada:

```
var expect = require('chai').expect;
var cervezas = require('./index');

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            // se comprueba que cumpla la condición de ser array de objetos
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            // se comprueba que incluya la cerveza Ambar
            done();
        });
    });
    describe('alazar', function () {
        it('Debería mostrar una cerveza de la lista', function (done) {
            //
            done();
        });
    });
});
```

- Por último realizamos los tests:

```
var expect = require('chai').expect;
var cervezas = require('./index');
var _ = require('lodash')

describe('cervezas', function () {
    describe('todas', function () {
        it('Debería ser un array de objetos', function (done) {
            expect(cervezas.todas).to.satisfy(isArrayOfObjects);
            function isArrayOfObjects(array){
                return array.every(function(item){
                    return typeof item === 'object';
                });
            }
            done();
        });
        it('Debería incluir la cerveza Ambar', function (done) {
            expect(cervezas.todas).to.satisfy(contieneAmbar);
            function contieneAmbar (array){
                return _.some(array, { 'nombre': 'ÁMBAR ESPECIAL' });
            }
            done();
        });

    });
    describe('alazar', function () {
        it('Debería mostrar un elemento de la lista de cervezas', function (done) {
            var cerveza = cervezas.alazar();
            expect(cervezas.todas).to.include(cerveza);
            done();
        });
    });
});
```

Automatizar tareas

- Cada vez que desarrollamos una versión de nuestra libería:

    - Ejecutar los tests
    - Hay que realizar un commit
    - Hay que realizar un tag del commig
    - Push a GitHub
    - Publicar en npm
    - ...
- Vamos a intentar automatizar todo:
    - **Semantic Release** para la gestión de versiones
    - **Travis** como CI (continuous integration)

Instalación Semantic Release

- Paso previo (en Ubuntu 14.04, si no fallaba la instalación):

```
sudo apt-get install libgnome-keyring-dev
```

- Instalación y configuración:

```
sudo npm i -g semantic-release-cli
semantic-release-cli setup
```

- **.travis.yml**: contiene la configuración de Travis
- Cambios en package.json:
    - Incluye un nuevo script (*semantic-release*)
    - Quita la versión
    - Añade la dependencia de desarrollo de Semantic Release

Versiones del software

- Utilizamos semantic versioning
- Semantic Release se ejecuta a través de Travis CI
- Travis CI se ejecuta al hacer un push (hay que configurarlo desde la web)
- Los commit tienen que seguir las reglas del equipo de Angular

Uso de commitizen

- **commitizen** que nos ayudará en la generación de los mensajes de los commit.
- La instalación, siguiendo su documentación:

```
sudo npm install commitizen -g
commitizen init cz-conventional-changelog --save-dev --save-exact
```

- Habrá que ejecutar **git cz** en vez de **git commit** para que los commits los gestione commitizen

Cambio de versión

- Vamos a comprobar nuestro entorno añadiendo una funcionalidad
- Si pedimos cervezas.alazar() queremos poder recibir más de una

- Los tests:

```
it('Debería mostrar varias cervezas de la lista', function (done) {
    var misCervezas = cervezas.alazar(3);
    expect(misCervezas).to.have.length(3);
    misCervezas.forEach(function(cerveza){
        expect(cervezas.todas).to.include(cerveza);
    });
    done();
});
```

- Añadimos la funcionalidad en el *src/index.js*: ``` var cervezas = require('./cervezas.json'); var uniqueRandomArray = require('unique-random-array'); var = *require('lodash'); var getCerveza = uniqueRandomArray(cervezas) module.exports = { todas:* .sortBy(cervezas, ['nombre']), alazar: alazar }

function alazar(unidades) { if (unidades===undefined){ return getCerveza(); } else { var misCervezas = []; for (var i = 0; i<unidades; i++) { misCervezas.push(getCerveza()); } return misCervezas; } }

```
- Hagamos ahora el git cz & git push y veamos como funciona todo
- Podríamos añadir un issue y hacer el fix en este commit escribiendo closes #issue en el footer del commit mes
sage.

## Git Hooks
- Son una manera de ejecutar scripts antes de que ocurra alguna acción
- Sería ideal pasar los tests antes de que se hiciera el commit
- Los Git Hooks son locales:
    - Si alguien hace un clone del repositorio, no tiene los GitHooks
    - Instalaremos un paquete de npm para hacer git hooks de forma universal
```

npm i -D ghooks

```
- Lo configuraremos en el package.json en base a la [documentación del paquete](https://www.npmjs.com/package/g
hooks):
```

"config": { "ghooks": { "pre-commit": "npm test" } }

```
## Coverage
- Nos interesa que todo nuestro código se pruebe mediante tests.
- Necesitamos una herramienta que compruebe el código mientras se realizan los tests:
```

npm i -D instanbul

```
- Modificaremos el script de tests en el package.json:
```

istanbul cover -x *.test.js _mocha -- -R spec src/index.test.js

```
- Instanbul analizará la cobertura de todos los ficheros excepto los de test ejecutando a su vez _mocha (un wra
pper de mocha proporcionado por ellos) con los tests.
- Si ejecutamos ahora *npm test* nos ofrecerá un resumen de la cobertura de nuestros tests.
- Por último nos crea una carpeta en el proyecto *coverage* donde podemos ver los datos, por ejemplo desde un n
avegador (fichero index.html)
- ¡Ojo, recordar poner la carpeta coverage en el .gitignore!

## Check coverage
- Podemos también evitar los commits si no hay un porcentaje de tests óptimo:
```

"pre-commit": "npm test && npm run check-coverage"

```
- Creamos el script check-coverage dentro del package.json:
```

"check-coverage": "istanbul check-coverage --statements 100 --branches 100 --functions 100 -lines 100"

```
- Podemos comprobar su ejecución desde el terminal mediante *npm run check-coverage* y añadir una función nueva
 sin tests, para comprobar que el check-coverage no termina con éxito.
- Lo podemos añadir también en Travis, de modo que no se haga una nueva release si no hay ciertos estándares (e
l test si lo hace por defecto):
```

script:

- npm run test
- npm run check-coverage ```

Gráficas

- Utilizaremos la herramienta codecov.io:

  ```
  npm i -D codecov.io
  ```

- Crearemos un script que recoge los datos de istanbul:

  ```
  "report-coverage": "cat ./coverage/lcov.info | codecov"
  ```

- Lo añadimos en travis de modo que genere un reporte:

  ```
  after success:
  - npm run report-coverage
  - npm run semantic-release
  ```

- Integrado con github (chrome extension)
- Por último podemos añadir etiquetas de muchos servicios: npm, codecov, travis... una fuente habitual es http://www.shields.io

adf

Instalación y configuración del software

nvm

- nvm solo está disponible en Linux/Mac. En Windows se puede usar nvm-windows
- Las instrucciones siguientes son para Linux
- Instalaremos y utilizaremos node vía nvm (node virtual manager)
- Esto nos permitirá:
- Poder cambiar de versión de node de forma transparente
- Evitar tener que hacer sudo cuando instalemos paquetes de forma global

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.1/install.sh | bash
```

- Instalar una versión de node:

```
nvm install 5.0
```

- Ver las versiones que hay instaladas:

```
nvm ls
```

- Usar una versión en particular:

```
nvm use 5.0
```

- Usar una versión en particular siempre que abrimos un shell:

```
nvm alias default 5.0
```

node

- La otra opción sería instalar directamente node:

```
curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo apt-get install build-essential
```

- En Windows utilizaremos esta opción descargando el paquete msi
- Comprobamos que esté instalado:

```
npm -v
node -v
```

Instalación de MongoDB

- Instalaremos primero mongodb:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.lis
t.d/mongodb-org-3.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
```

- El servicio se levanta como otros servicios de Linux:

  ```
  sudo service mongod start
  ```

- Y para entrar a su consola, mediante **mongo**, o mediante algún gui como por ejemplo Robomongo, que también podemos instalar desde su web.

Editor de código

- Utilizaremos Visual Code Editor
- Es un producto open source de Microsoft realizado mediante node.js (electron)
- Uso muy similar a Sublime Text
- Tiene un excepcional debugger para node.js

Debug en node.js

Programa de ejemplo

- Vamos a utilizar un ejemplo muy sencillo que nos sirva para:
  - Aprender a hacer debug
  - Conocer como funcionan los módulos en nodejs
  - Familiarizarnos con el editor de código
- Fichero suma.js

```
let suma = function(a, b) {
return a+b;
}
module.exports = suma;
```

- La función suma la queremos utilizar desde nuestro programa, por eso lleva el *module.exports*, que indica las partes de este módulo (fichero) que se pueden exportar (utilizar desde otro fichero).
- Mi aplicación (cargo las librerías que necesito mediante require y las utilizo)

```
let suma = require('./suma.js');
console.log (suma(3,5));
```

Opciones de debug

- Utilizando la consola
- Utilizando Chrome Developer Tools
- Mediante Visual Studio

Debug en consola

- Ejecutamos el siguiente comando

```
node debug app.js
```

- Podemos incluir puntos de interrrupción añadiendo líneas con la instrucción :

```
debugger;
```

- Las teclas de acceso rápido son:
  - Hasta el siguiente breackpoint: c
  - Hasta la siguiente línea de código en el mismo fichero: n
  - Entrar en función: s
  - Salir de función: o

Mediante Chrome Developer Tools

- Hay dos opciones dependiendo de la versión de node:

    - Mediante el propio node

```
node --inspect app.js
```

        - Experimental en v6.x
    - Mediante un paquete adicional, **node-inspector**
        - Deprecated en v7.x
- Mediante el comando **node --inspect**

```
⇒  node --inspect app.js

Debugger listening on port 9229.

Warning: This is an experimental feature and could change at any time.

To start debugging, open the following URL in Chrome:

    chrome-devtools://devtools/remote/serve\_file/@60cd6e859b9f557d2312f5bf532f6aec5f284980/inspector.html?expe
riments=true&v8only=true&ws=127.0.0.1:9229/bb7de882-abe4-4c61-8099-22908239de18
```

- Se suele poner un punto de interrupción al empezar:

```
node --debug-brk --inspect app.js
```

- El enlace que nos proporciona se abre con el navegador Chrome y se puede inspeccionar el código mediante sus herramientas de desarrollo.
- Instalo el paquete node-inspector:

```
npm i -g node-inspector
```

- Ejecuto el ejecutable node-debug:

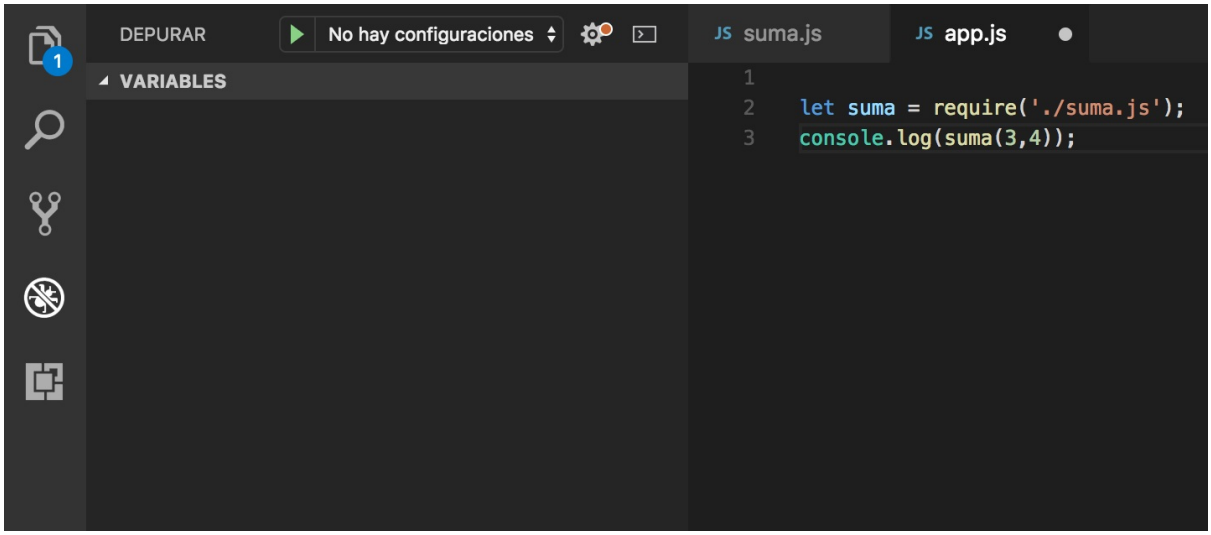
```
node-debug app.js
```

Debug en Visual Studio Code

- Primero hay que ir a la vista de debug, pulsando el icono correspondiente:

visual studio code debugger

- Para hacer debug necesitamos un fichero de configuración *launch.json*, que se obtiene pulsandos en la ruleta



- Ya podemos hacer debug como en cualquier otro programa:
  - Play para empezar
  - Break points pulsando a la izquierda de la numeración de lineas del código
  - Puedes pulsar con el botón derecho y poner breakpoints condicionales.
- Buen tutorial