Node.js

Introducción

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)

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

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

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

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 message.

## 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/ghooks>):
```

"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 wrapper 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 navegador (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 (el 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

Arquitectura API REST

Creación de una API

Qué es una API

- Es una forma de describir la forma en que los programas o los sitios webs intercambian datos.
- El formato de intercambio de datos normalmente es **JSON** o XML.

¿Para qué necesitamos una API?

- Ofrecer datos a aplicaciones que se ejecutan en un movil
- Ofrecer datos a otros desarrolladores con un formato más o menos estándar.
- Ofrecer datos a nuestra propia web/aplicación
- **Consumir datos** de otras aplicaciones o sitios Web

Provedores de APIs

- Algunos ejemplos de sitios web que proveen de APIS son:
  - Twitter: acceso a datos de usuarios, estado
  - Google: por ejemplo para consumir un mapa de Google
- Pero hay muchos más: Facebook, YouTube, Amazon, foursquare...
- Pero todavía hay muchos más: directorio de proveedores de APIs

Qué significa API REST

- REST viene de, **RE**presentational **S**tate **T**ransfer
- Es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.
- REST se compone de una lista de reglas que se deben cumplir en el diseño de la arquitectura de una API.
- Hablaremos de **servicios web restful** si cumplen la arquitectura REST.
- Restful = adjetivo, Rest = Nombre

Como funciona REST

Llamadas al API

- Las llamadas al API se implementan como peticiones HTTP, en las que:
  - La URL representa el **recurso**

```
<http://www.formandome.es/api/cursos/1>
```

- El método (HTTP Verbs) representa la **operación**:

```
GET <http://www.formandome.es/api/cursos/1>
```

- El código de estado HTTP representa el **resultado**:

```
200 OK HTTP/1.1
404 NOT FOUND HTTP/1.1
```

Creación de recursos

- La URL estará “abierta” (el recurso todavía no existe y por tanto no tiene id)
- El método debe ser POST

```
<http://eventos.com/api/eventos/3/comentarios>
```

Respuesta a la creación de recursos

- Resultados posibles:
  - 403 (Acceso prohibido)
  - 400 (petición incorrecta, p.ej. falta un campo o su valor no es válido)

    - 500 (Error del lado del servidor al intentar crear el recurso, p.ej. se ha caído la BD)
    - 201 (Recurso creado correctamente)
- ¿Qué URL tiene el recurso recién creado?
    - La convención en REST es devolverla en la respuesta como valor de la cabecera HTTP Location

Actualización de recursos

- Método PUT
    - Según la ortodoxia REST, actualizar significaría cambiar TODOS los datos
    - PATCH es un nuevo método estándar HTTP (2010) pensado para cambiar solo ciertos datos. Muchos frameworks de programación REST todavía no lo soportan
- Resultados posibles
    - Errores ya vistos con POST
    - 201 (Recurso creado, cuando le pasamos el id deseado al servidor)
    - 200 (Recurso modificado correctamente)

Eliminar recursos

- Método DELETE
- Algunos resultados posibles:
    - 200 OK
    - 404 Not found
    - 500 Server error
- Tras ejecutar el DELETE con éxito, las siguientes peticiones GET a la URL del recurso deberían devolver 404

Arquitectura REST

Reglas de una arquitectura REST

- Interfaz uniforme
- Peticiones sin estado
- Cacheable
- Separación de cliente y servidor
- Sistema de Capas
- Código bajo demanda (opcional)

Interfaz Uniforme

- La interfaz de basa en recursos (por ejemplo el recurso Empleado (Id, Nombre, Apellido, Puesto, Sueldo)
- El servidor mandará los datos (vía html, json, xml...) pero lo que tenga en su interior (BBDD por ejemplo) para el cliente es transparente
- La representación del recurso que le llega al cliente, será suficiente para poder cambiar/borrar el recurso:
  - Suponiendo que tenga permisos
  - Por eso en el recurso solicitado se suele enviar un parámetro Id

Interfaz uniforme: mensajes descriptivos

- Mensajes descriptivos:
  - Usar las características del protocolo http para mejorar la semántica:
    - HTTP Verbs
    - HTTP Status Codes
    - HTTP Authentication
  - Procurar una API sencilla y jerárquica y con ciertas reglas: **uso de nombres en plural**

Peticiones sin estado

- http es un protocolo sin estado ---> mayor rendimiento

```
GET mi_url/empleados/1234
DELETE mi_url/empleados/1234
```

- En la segunda petición hemos tenido que incidar el identificador del recurso que queremos borrar
- El servidor no guardaba los datos de la consulta previa que tenía el cliente en partícular.
- Una petición del tipo *DELETE mi_url/empleado* debe dar error, ¡falta el id y el servidor no lo conoce!

Cacheable

- En la web los clientes pueden cachear las respuestas del servidor
- Las respuestas se deben marcar de forma implícita o explícita como cacheables o no.

- En futuras peticiones, el cliente sabrá si puede reutilizar o no los datos que ya ha obtenido.
- Si ahorramos peticiones, mejoraremos la escabilidad de la aplicación y el rendimiento en cliente (evitamos principalmente la latencia).

Separación de cliente y servidor

- El cliente y servidor están separados, su unión es mediante la interfaz uniforme
- Los desarrollos en frontend y backend se hacen por separado, teniendo en cuenta la API.
- Mientras la interfaz no cambie, podremos cambiar el cliente o el servidor sin problemas.

Sistema de capas

- El cliente puede estar conectado mediante la interfaz al servidor o a un intermediario, para el es irrelevante y desconocido.
- Al cliente solo le preocupa que la API REST funcione como debe: no importa el COMO sino el QUE
- El uso de capas o servidores intermedios puede servir para aumentar la escalabilidad (sistemas de balanceo de carga, cachés) o para implementar políticas de seguridad

Código bajo demanda (opcional)

- Los servidores pueden ser capaces de aumentar o definir cierta funcionalidad en el cliente transfiriéndole cierta lógica que pueda ejecutar:
  - Componentes compilados como applets de Java
  - JavaScript en cliente.

Consejos para elaborar una API REST

Versiones del API

- Los cambios en el código no deberían afectar al API
- Si hay cambios en el API se deben usar versiones para no fustrar a los desarrolladores
- La mejor opción es añadir un prefijo a las URLS:

```
GET /v1/geocode HTTP/1.1
Host: api.geocod.io

GET /v2/geocode HTTP/1.1
Host: api.geocod.io
```

HTTP verbs

- Si realizamos CRUD, debemos utilizar los HTTP verbs de forma adecuada para cuidar la semántica.
  - *GET*: Obtener datos. Ej: GET /v1/empleados/1234
  - *PUT*: Actualizar datos. Ej: PUT /v1/empleados/1234
  - *POST*: Crear un nuevo recurso. Ej: POST /v1/empleados
  - *DELETE*: Borrar el recurso. Ej: DELETE /v1/empleados/1234
  - *¿PATCH?*: Para actualizar ciertos datos

Nombre de los recursos

- Plural mejor que singular, para lograr uniformidad:
  - Obtenemos un listado de clientes: GET /v1/clientes
  - Obtenemos un cliente en partícular: GET /v1/clientes/1234
- Url's lo más cortas posibles
- Evita guiones y guiones bajos
- Deben ser semánticas para el cliente
- Utiliza nombres y no verbos
- Estructura jerárquica para indicar la estructura: /v1/clientes/1234/pedidos/203

Códigos de estado

- Se utilizan los códigos de estado de http
- Si realizamos un request de POST deberemos devolver un 201.
- Se pueden producir múltiples errores en la llamada al API:
  - falta de permisos
  - errores de validación
  - O incluso un error interno de servidor.
- Siempre se debe devolver un código de estado HTTP con los requests.
- Añadir un mensaje de error si es necesario.

Formato de salida

- En función de la petición nuestra API podría devolver uno u otro formato.
- Nos fijaremos en el ACCEPT HEADER

```
GET /v1/geocode HTTP/1.1
Host: api.geocod.io
Accept: application/json

*GET /v1/geocode HTTP/1.1
Host: api.geocod.io
Accept: application/xml
```

- En principio utilizaremos JSON: sencillo y simple
- XML no es nuestro amigo: schemas, namespaces...
- Si no es un requerimiento, evitaremos XML

Solicitudes AJAX entre dominios

Problemática

- El modelo de seguridad de las aplicaciones web no permite en principio realizar peticiones Ajax entre dominios
- En general esto es OK, a nadie le gustaría que código malicioso de una URL estuviera accediendo a ningún dato de la solapa del navegador que está abierta con nuestra cuenta bancaria
- Pero también es un problema, por ejemplo con AJAX: una página de http://localhost en principio no puede hacer una petición AJAX a Google
- Como consecuencia, se han tenido que idear diversos “trucos”/técnicas para intentar sobrepasar este límite

Técnicas para evitar las restricciones de seguridad

- Lo más habitual es el uso de JSONP (JSON con Padding)
- Se basa en que aunque no podemos consumir datos de otro dominio vía XHR, si podemos cargar un script de dicho dominio

CORS

- En el 2008 se publicó la primera versión de la especificación XMLHttpRequest Level 2.
- CORS es el acrónimo de Cross-origin resource sharing.
- Debemos tener presente, que los navegadores antiguos no soportan CORS.
- Esta nueva especificación, añade funcionalidades nuevas a las peticiones AJAX como las peticiones entre dominios (cross-site), eventos de progreso y envio de datos binarios.
- CORS requiere configuraciones en el servidor.

Autenticación y validación en API REST

Métodos de autenticación

- Básicamente tenemos dos formas de implementar la autenticación en servidor:
  - **Basada en cookies**, la más utilizada:
    - El servidor guarda la cookie para autenticar al usuario en cada request.
    - Habrá que tener un almacen de sesiones: en bbdd, Redis...

```
- **Basada en tokens**, se confía en un token firmado que se envía al servidor en cada petición
```

Image not found or type unknown https://juanda.gitbooks.io/webapps/content/api/cookie-token-auth.png

¿Qué es un token?

- Un token es un valor que nos autentica en el servidor
  - Normalmente se consigue después de hacer login mediante usuario/contraseña
- ¿Cómo se genera el token?
  - Normalmente un hash calculado con algún dato (p.ej. login del usuario + clave secreta)
  - Además el token puede llevar datos adicionales como el login
- ¿Cómo comprueba el servidor que es válido?
  - Generando de nuevo el Hash y comprobando si es igual que el que envía el usuario (100% stateless)
  - O bien habiendo almacenado el Hash en una B.D. asociado al usuario y simplemente comprobando que coincide

Beneficios de usar tokens

- Uso entre dominios
  - cookies + CORS no se llevan bien

OAuth

- Para que una app pueda acceder a servicios de terceros sin que el usuario tenga que darle a la app sus credenciales del servicio
  - Ejemplo: una app que permite publicar en tu muro de FB, pero en la que no confías lo suficiente como para meter tu login y password de FB
- Es el estándar en APIs REST abiertos a terceros
- Se basa en el uso de un token de sesión

Terminología de OAuth

- En un proceso AUTH intervienen 3 actores:
  - **Consumer:** El servicio al que el usuario quiere acceder usando una cuenta externa.
  - **Service Provider:** Al servicio de autenticación externo se le llama Service Provider.
  - **El usuario final**

Flujo en OAuth

- El consumer pide un token al service provider, esto es transparente para el usuario.
- El consumer redirige al usuario a una página segura en el service provider, pasándole el token como parámetro.
- El usuario se autentica en la página del service provider, validando el token.
- El service provider envía al usuario de vuelta a la página del consumer especificada en el parámetro oauth_callback.
- El consumer recoge al usuario en la callback URL junto con el token de confirmación de identidad.

Creación de una API

Creación de una API con node.js

Primeros pasos

- npm configurado y git configurado
- Crear repositorio en GitHub
- Clonar repositorio
- Ejecutar npm init
- Crear la estructura de la aplicación, por el momento una carpeta app donde iremos guardando el código propio de la aplicación

Configuración de eslint

- Si tienes los linters para Sublime y js configurados, comprueba que la consola arroja un error.
- Hay que configurar eslint para formatear nuestro código.

```
npm i -D eslint
```

- Como lo instalo en local, para ejecutarlo, necesito darle el path:

```
./node_modules/.bin/eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? No
? Where will your code run? Node
? Do you use JSX? No
? What style of indentation do you use? Spaces
? What quotes do you use for strings? Single
```

```
? What line endings do you use? Unix
? Do you require semicolons? No
? What format do you want your config file to be in? JSON
Successfully created .eslintrc.json file in /home/juanda/api_node_express/ejercicio3-nodemon-eslint
```

- Comprueba que el linter de JavaScript te funcione bien

express

- Utilizaremos express para realizar la API
- Instalar express mediante uno de los comandos siguientes:

```
npm install --save express
npm i -S express
```

- Creamos el fichero app/server.js donde pondremos el código necesario para testear una API muy básica para probar Express.

https://www.guru99.com/images/NodeJS/010716_0613_NodejsExpre1.png
Image not found or type unknown

- Utiliza el plugin ExpressComplete (aget, aput...) de Sublime para autocompletado

```
var express = require('express') //llamamos a Express
var app = express()

var port = process.env.PORT || 8080  // establecemos nuestro puerto

app.get('/', function(req, res) {
  res.json({ mensaje: '¡Hola Mundo!' })
})

app.get('/cervezas', function(req, res) {
  res.json({ mensaje: '¡A beber cerveza!' })
})

app.post('/', function(req, res) {
  res.json({ mensaje: 'Método post' })
})
```

```
app.del('/', function(req, res) {
  res.json({ mensaje: 'Método delete' })
})

// iniciamos nuestro servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)
```

- Si el linter de js te da un error por usar *console.log*, puedes deshabilitar esa regla en el fichero de configuración de eslint

```
no-console: 0
```

Iniciar y testear a mano nuestra API

- Iniciamos nuestro API Server mediante el comando

```
node app/server.js
```

- Probamos que la API funcione mediante http://localhost:8080 o utilizando la extensión de Google Chrome **Postman**
- Creamos una entrada en nuestro fichero package.json, de modo que podamos arrancar nuestra API mediante `npm start`

```
"start": "node app/server.js"
```

- Hagamos un commit del repositorio, pero sin tener en cuenta la carpeta node_modules.
- Comprueba con el segundo *git status* que git no lo tiene en cuenta antes de continuar con el commit:

```
git status
echo "node_modules">.gitignore
git status
git add -A *
git commit -m "Primera versión API"
git push
```

- Debemos hacer nuevas instantáneas en pasos posteriores, pero ya no las documentaré por brevedad.

nodemon

- Es un wrapper de node, para reiniciar nuestro API Server cada vez que detecte modificaciones.

```
npm i -D nodemon
```

- Cada vez que ejecutemos **npm start** ejecutaremos nodemon en vez de node. Habrá que cambiar el script en el fichero *package.json*:

```
"start": "nodemon app/server.js"
```

Uso de enrutadores

- Imagina que nuestra API es compleja:
  - Tiene varios recursos (GET, POST... por cada uno de ellos)
  - Versionado de la API
- Utilizaremos enrutadores
  - Asociamos enrutadores a la app en vez de rutas directamente
  - Cada enrutador se puede asociar por ejemplo a un recurso
  - Se pueden anidar enrutadores
- El código para un enrutador sería así:

```
// para establecer las distintas rutas, necesitamos instanciar el express router
var router = express.Router()

//establecemos nuestra primera ruta, mediante get.
router.get('/', function(req, res) {
  res.json({ mensaje: '¡Bienvenido a nuestra API!' })
})

// nuestra ruta irá en <http://localhost:8080/api>
// es bueno que haya un prefijo, sobre todo por el tema de versiones de la API
app.use('/api', router)
```

Recibir parámetros

- Cuando el router recibe una petición, podemos observar que ejecuta una función de callback:

```
function (req, res){}
```

  - El parámetro **req** representa la petición (request)
  - El parámetro **res** representa la respuesta (response)
  - En el caso anterior hemos codificado la respuesta en json:

```
res.json({ mensaje: '¡Bienvenido a nuestra API!' })
```

- A menudo la petición se hará enviando algún parámetro adicional. Hay varias posibilidades:
  - Mediante la url: se recogerán mediante

```
req.param.nombreVariable
```

  - Mediante post en http hay dos posiblidades:
    - application/x-www-form-urlencoded
    - multipart/form-data
- En peticiones post, escogeremos x-www-form-urlencoded si no se envía una gran cantidad de datos (normalmente ficheros)

Parámetros por url

- Vamos a mandar un parámetro *nombre* a nuestra api, de modo que nos de un saludo personalizado.

```
router.get('/:nombre', function(req, res) {
  res.json({ mensaje: '¡Hola' + req.params.nombre })
})
```

Parámetros por post

- Parámetros mediante **POST y application/x-www-form-urlencoded**:
  - Necesitaremos body-parser: extrae los datos del body y los convierte en json
- Parámetros mediante **POST y multipart/form-data**
  - Usaremos https://www.npmjs.com/package/busboy o Multer

Ejemplo con body-parser

- Hay que instalar body-parser

```
npm i -S body-parser
```

- body-parser actúa como **middleware**
- El código adicional será similar al siguiente:

```
var bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

router.post('/',function(req,res) {
  res.json({mensaje: req.body.nombre})
})
```

Rutas de nuestra API

- Las rutas que utilizaremos son las siguientes:

Ruta	Verbo http	Descripción
/api/cervezas	GET	Obtenemos todas las cervezas
/api/cervezas/search?q=keyword	GET	Obtenemos cervezas por keyword
/api/cervezas/:id	GET	Obtenemos los datos de una cerveza
/api/cervezas	POST	Damos de alta una cerveza
/api/cervezas/:id	PUT	Actualizamos los datos de una cerveza
/api/cervezas/:id	DELETE	Borramos los datos de una cerveza

- Como la configuración se complica, lo suyo es dividirla:
  - Creamos una carpeta app/routes donde irán las rutas
    - Creamos un fichero *app/routes/index.js* donde irá el enrutador del versionado de la API
    - Creamos un fichero en *app/routes* por cada resouce (en este caso solo uno, *cervezas.js*)
- El fichero *app.js* queda así:

```
var express = require('express') //llamamos a Express
var app = express()
var bodyParser = require('body-parser')

var port = process.env.PORT || 8080  // establecemos nuestro puerto

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

// nuestra ruta irá en <http://localhost:8080/api>
// es bueno que haya un prefijo, sobre todo por el tema de versiones de la API
var router = require('./routes')
app.use('/api', router)

//arrancamos el servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)
```

- El fichero *app/routes/index.js*

```
var router = require('express').Router()
var cervezas = require('./cervezas')

router.use('/cervezas', cervezas)

router.get('/', function (req, res) {
  res.status(200).json({ message: 'Estás conectado a nuestra API' })
})

module.exports = router
```

- Y el fichero *app/routes/cervezas.js*:

```
var router = require('express').Router()
router.get('/search', function(req, res) {
  res.json({ message: 'Vas a buscar una cerveza' })
})
router.get('/', function(req, res) {
  res.json({ message: 'Estás conectado a la API. Recurso: cervezas' })
```

```
})
router.get('/:id', function(req, res) {
  res.json({ message: 'Vas a obtener la cerveza con id ' + req.params.id })
})
router.post('/', function(req, res) {
  res.json({ message: 'Vas a añadir una cerveza' })
})
router.put('/:id', function(req, res) {
  res.json({ message: 'Vas a actualizar la cerveza con id ' + req.params.id })
})
router.delete('/:id', function(req, res) {
  res.json({ message: 'Vas a borrar la cerveza con id ' + req.params.id})
})
module.exports = router
```

Acceso a base de datos

- Para la persistencia de nuestros datos utilizaremos una base de datos
- Optamos por una base de datos MongoDB:
  - Es lo más habitual en arquitecturas MEAN
  - Así operamos con objetos json tanto en node como en bbdd (bson)
  - Nos permite más libertad, al utilizar colecciones en vez de tablas

Instalación de MongoDB

- Instalamos y leventamos el servicio de 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.list.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
- La consola de Mongo también es un intérprete de JavaScript :-)

Inserción de datos

- Utilizaremos el fichero *cervezas.json*, lo podemos obtener mediante:

```
wget
<https://raw.githubusercontent.com/juanda99/proyecto_web_basica/master/cervezas.json>
```

- Importar nuestro cervezas.json a una base de datos

```
mongoimport --db web --collection cervezas --drop --file cervezas.json --jsonArray
```

- Para poder hacer una búsqueda por varios campos de texto, tendremos que hacer un índice:

```
$ mongo # para entrar en la consola de mongo
use web; #seleccionamos la bbdd
db.cervezas.createIndex(
 {
   "Nombre": "text",
   "Descripción": "text"
 },
 {
   name: "CervezasIndex",
   default_language: "spanish"
 }
)
```

- Comprobamos que el índice esté bien creado

```
db.cervezas.getIndexes()
```

- Si hiciera falta, lo podemos recrear:

```
db.cervezas.dropIndex("CervezasIndex")
```

Instalación de Mongoose

- Instalaremos [mongoose] como ODM (Object Document Mapper) en vez de trabajar con el driver nativo de MongoDB (se utiliza por debajo).

```
npm i -S mongoose
```

Uso de Mongoose

- Creamos el fichero *app/db.js* donde configuraremos nuestra conexión a base de datos mediante mongoose:

```
//incluimos Mongoose y abrimos una conexión
var mongoose = require('mongoose')
var MONGO_URL = process.env.MONGO_URL || 'mongodb://localhost/web'
mongoose.connect(MONGO_URL)

mongoose.connection.on('connected', function () {
  console.log('Conectado a la base de datos: ' + MONGO_URL)
})

mongoose.connection.on('error',function (err) {
  console.log('Error al conextar a la base de datos: ' + err)
})

mongoose.connection.on('disconnected', function () {
  console.log('Desconectado de la base de datos')
})

process.on('SIGINT', function() {
  mongoose.connection.close(function () {
    console.log('Desconectado de la base de datos al terminar la app')
    process.exit(0)
  })
})
```

- En nuestro fichero *app/server.js* incluimos el fichero de configuración de bbdd:

```
var express = require('express') //llamamos a Express
var bodyParser = require('body-parser')
/*toda la configuración de bbdd la hacemos en un fichero a parte*/
```

```
require('./db')
....
```

Modelos

- Definimos un esquema para nuestros objetos (cervezas) y creamos nuestro modelo a partir del esquema (fichero *app/models/Cervezas.js*):

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

var cervezaSchema = new Schema({
  Nombre: String,
  Descripción: String,
  Graduacion: String,
  Envase: String,
  Precio: String
})

var Cerveza = mongoose.model('Cerveza', cervezaSchema)

module.exports = Cerveza
```

- Ahora podríamos crear documentos y guardarlos en la base de datos:

```
var miCerveza = new Cerveza({ name: 'Ambar' });
miCerveza.save(function (err, miCerveza) {
  if (err) return console.error(err);
  console.log('Guardada en bbdd' + miCerveza.name);
})
```

Uso de controladores

- Desde nuestro fichero de rutas (*app/routes/cervezas.js*), llamaremos a un controlador que será el encargado de añadir, borrar o modificar cervezas en base al modelo anterior.
- Nuestro código queda así perfectamente separado y cada fichero de rutas se encargará solo de gestionar los endpoints de nuestra API para el recurso en cuestión.

- Creamos un directorio específico para los controladores, donde colocaremos el específico para las cervezas.
- Nuestro código quedas así (fichero *app/controllers/cervezasController.js*):

```
var Cervezas = require('../models/Cervezas')
module.exports = {
// <https://docs.mongodb.com/v3.0/reference/operator/query/text/>
search: function (req, res) {
  var q = req.query.q
  Cervezas.find({ $text: { $search: q } }, function(err, cervezas) {
    if(err) {
      return res.status(500).json({
        message: 'Error en la búsqueda'
      })
    }
    return res.json(cervezas)
  })
},
list: function(req, res) {
  Cervezas.find(function(err, cervezas){
    if(err) {
      return res.status(500).json({
        message: 'Error obteniendo la cerveza'
      })
    }
    return res.json(cervezas)
  })
},
show: function(req, res) {
  var id = req.params.id
  Cervezas.findOne({_id: id}, function(err, cerveza){
    if(err) {
      return res.status(500).json({
        message: 'Se ha producido un error al obtener la cerveza'
      })
    }
    if(!cerveza) {
      return res.status(404).json( {
        message: 'No tenemos esta cerveza'
      })
```

```
    }
    return res.json(cerveza)
  })
},
create: function(req, res) {
  var cerveza = new Cervezas (req.body)
  cerveza.save(function(err, cerveza){
    if(err) {
      return res.status(500).json( {
        message: 'Error al guardar la cerveza',
        error: err
      })
    }
    return res.status(201).json({
      message: 'saved',
      _id: cerveza._id
    })
  })
},
update: function(req, res) {
  var id = req.params.id
  Cervezas.findOne({_id: id}, function(err, cerveza){
    if(err) {
      return res.status(500).json({
        message: 'Se ha producido un error al guardar la cerveza',
        error: err
      })
    }
    if(!cerveza) {
      return res.status(404).json({
        message: 'No hemos encontrado la cerveza'
      })
    }
    cerveza.Nombre = req.body.nombre
    cerveza.Descripción =  req.body.descripcion
    cerveza.Graduacion = req.body.graduacion
    cerveza.Envase = req.body.envase
    cerveza.Precio = req.body.precio
    cerveza.save(function(err, cerveza){
```

```
      if(err) {
        return res.status(500).json({
          message: 'Error al guardar la cerveza'
        })
      }
      if(!cerveza) {
        return res.status(404).json({
          message: 'No hemos encontrado la cerveza'
        })
      }
      return res.json(cerveza)
    })
  })
},
remove: function(req, res) {
  var id = req.params.id
  Cervezas.findByIdAndRemove(id, function(err, cerveza){
    if(err) {
      return res.json(500, {
        message: 'No hemos encontrado la cerveza'
      })
    }
    return res.json(cerveza)
  })
}
}
```

- El router que gestiona el recurso se encarga de llamarlo (fichero *app/routes/cervezas.js*):

```
var router = require('express').Router()
var cervezasController = require ('../controllers/cervezasController')

router.get('/search', function(req, res) {
  cervezasController.search(req, res)
})
router.get('/', function(req, res) {
  cervezasController.list(req, res)
})
router.get('/:id', function(req, res) {
  cervezasController.show(req, res)
```

```
})
router.post('/', function(req, res) {
  cervezasController.create(req, res)
})
router.put('/:id', function(req, res) {
  cervezasController.update(req, res)
})
router.delete('/:id', function(req, res) {
  cervezasController.remove(req, res)
})
module.exports = router
```

Test desde el navegador o mediante Postman

- Comprobamos que se genera el listado de cervezas
- Comprobamos que se busca por keyword:...

```
<http://localhost:8080/api/cervezas/search?q=regaliz>
```

Test de la API

- Utilizaremos Mocha como test framework y supertest para hacer las peticiones http.

```
npm i -D mocha supertest
```

- Creamos nuestro fichero *tests/api.test.js* con la prueba para crear una cerveza:

```
'use strict'
/* global describe it */
var request = require('supertest')

/*obtenemos nuestra api rest que vamos a testear*/
var app = require('../app/server')

describe('Crear una nueva cerveza', function() {
```

```
  it('Crea la cerveza retornando 201', function(done) {
    request(app)
      .post('/api/cervezas/')
      .set('Accept', 'application/json')
      .expect('Content-Type', /json/)
      .send({
        'Nombre': 'DAMN',
        'Descripción': 'Mi cerveza preferida',
        'Graduación': '10º',
        'Envase': 'Bidón',
        'Precio': '1 eurito'
      })
      .expect(201, done)
  })
})
```

- Utiliza los paquetes **Mocha Snippets** y **Chai Completions** de Sublime Text para completar el código
- Si echamos un vistazo al código anterior:
- **describe** nos sirve para describir los test suites (se pueden anidar varios)
  - **it** nos sirve para describir cada caso de test.
  - **requests(app).post** realizará una petición post a nuestra api
- ¿Cómo ejecutamos el test?
  - Mediante línea de comandos (hace falta la ruta completa ya que no hemos instalado el paquete de forma global):

```
node_modules/.bin/mocha tests
```

  - Introduciendolo en el fichero package.json (no hace falta la ruta al estar dentro del package.json):

```
"test": "mocha tests"
```

- Para que el caso anterior funcione, tenemos que modificar nuestro fichero *app/server* de modo que se pueda utilizar *app* desde otro fichero js:

```
/*lo añado al final de app/server.js:*/
module.exports = app
```

- Por último podríamos utilizar un paquete como **istanbul** que nos analice el código y ver si nuestras pruebas recorren todas las instrucciones, funciones o ramas del código:

```
npm i -D istanbul
./node_modules/.bin/istanbul cover -x "**/tests/**"  ./node_modules/.bin/_mocha
tests/api.test.js
```

- Estos datos son facilmente exportables a algún servicio que nos de una estadística de la cobertura de nuestros tests o que haga un seguimiento de los mismos entre las distintas versiones de nuestro código.
- Por último también se podría integrar con un sistema de integración continua tipo Travis.

Uso de middlewares

- Son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación.
- La siguiente función de middleware se denota normalmente con una variable denominada next.
- Podemos crear un middleware que guarde traza de las fechas de acceso:

```
var app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

- Normalmente utilizaremos middlewares que ya están hechos, por ejemplo Morgan para logs y cors para Cors.
- Los instalamos:

```
npm i -S cors morgan
```

- Los insertamos en nuestra API (el orden puede ser importante):

```
var express = require('express') //llamamos a Express
var app = express()
var cors = require('cors')
var bodyParser = require('body-parser')
var morgan = require('morgan')

var port = process.env.PORT || 8080  // establecemos nuestro puerto

/*toda la configuración de bbdd la hacemos en un fichero a parte*/
require('./db')
```

```
app.use(morgan('combined'))
app.use(cors())
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

// para establecer las distintas rutas, necesitamos instanciar el express router
var router = require('./routes')
app.use('/api', router)

// iniciamos nuestro servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)

/*lo añado al final de app/server.js:*/
module.exports = app
```