



Escuela de Ingeniería Informática

Escuela de Ingeniería Informática
School of Computer Science Engineering

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Sistemas Distribuidos e Internet

Tema 7 - Parte 2 Introducción a Node.js



Dr. Edward Rolando Núñez Valdez

nunezedward@uniovi.es

Índice

- Arquitectura y módulos
- Bases de datos MongoDB
 - Servidor de bases de datos MongoDB
 - Conexión a bases de datos MongoDB
 - Gestión de datos con MongoDB
 - Arquitectura: acceso a datos
 - Insert
 - Remove
 - Update
 - Find
 - ObjectID

Índice

- Encriptación (Cifrado)
- Autenticación y autorización
 - Autenticación
 - Sesión y autorización
 - Enrutadores
- Subida de ficheros
- Captura de errores
- HTTPS

Arquitectura y módulos

- La aplicación Node.js debería seguir una **arquitectura modular**
 - Dividendo las responsabilidades en diferentes módulos
 - Estos módulos pueden comunicarse entre sí
- Comparándolo con Spring podríamos decir que:
 - Los módulos que definen endpoints o URLs actúan como **controladores**.
 - Los módulos que definen lógica de negocio actúan como **servicios**
 - Los módulos de acceso a datos, actúan como **repositorios**.

Arquitectura y módulos

- Por sus características Node.js es un muy buen candidato para entornos **muy dinámicos y desarrollos ágiles**
 - Aplicaciones con requerimientos que se modifican frecuentemente o han sido poco definidos
 - Los cambios deben realizarse de forma rápida y efectiva (modificando poco código)
- En un desarrollo rápido el tamaño de la aplicación puede condicionar su arquitectura
 - Algunas aplicaciones con poca lógica de negocio pueden incluso prescindir de la capa de **servicios** (sí estos servicios realizan básicamente llamadas a repositorios)

Arquitectura y módulos

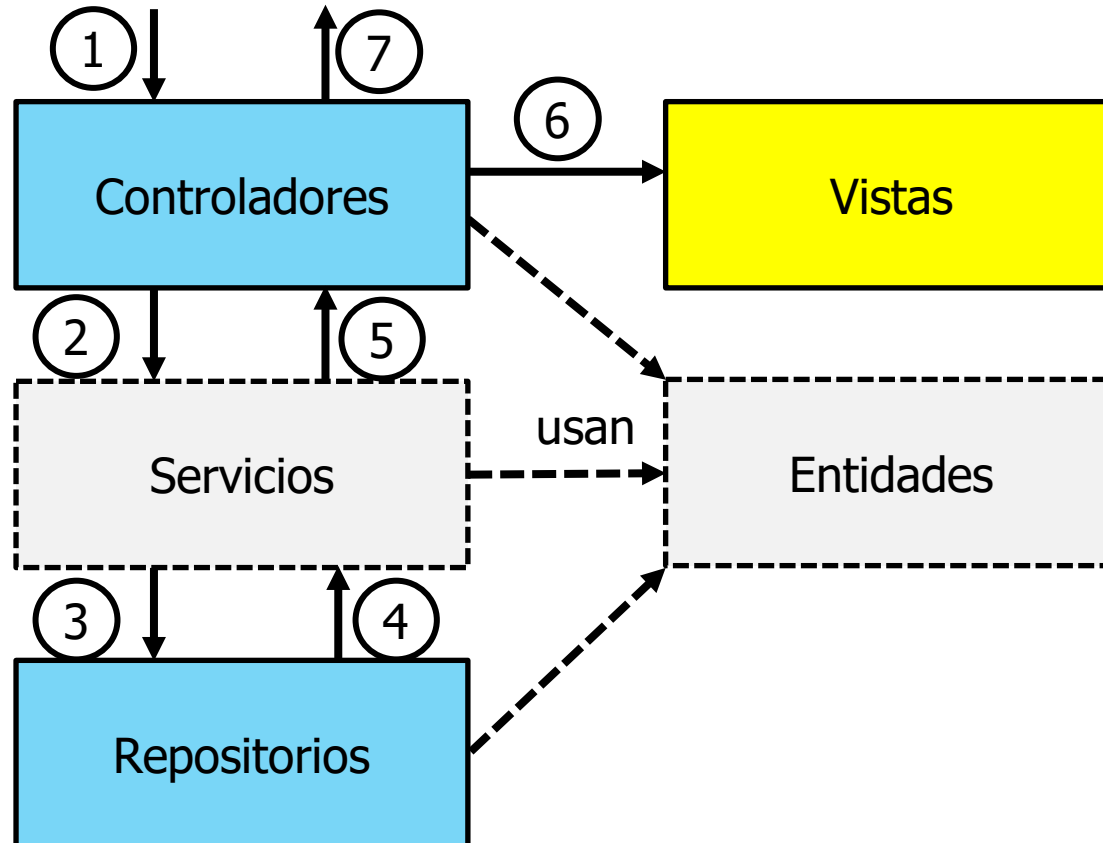
- Muchas aplicaciones no formalizan las entidades de forma estricta (con clases)
 - Una alternativa es usar **objetos**, donde resulta muy rápido modificar/añadir campos
 - Por ejemplo: objeto genérico canción (song) con 3 atributos

```
var song = {  
  title : req.body. title,  
  kind  : req.body. kind,  
  price : req.body.price  
}
```

- Los objetos se pueden utilizar de forma directa en muchas bases de datos no relacionales

Arquitectura y módulos


- La arquitectura de una aplicación pequeña y dinámica podría prescindir de la capa de **servicios** y **entidades** (clases)



DEFINICIÓN DE MÓDULOS

Arquitectura y módulos

- Los módulos pueden definir una *función*, *objeto* o *clase*
 - Dependiendo del objetivo se opta por uno u otro
- Los módulos que declaran una **función** ejecutan la función al ser incluidos (**require(<módulo>)**)
 - Ejemplo: módulos que agregan rutas a la aplicación



```
module.exports = function(app, swig) {  
  app.get("/publications", function(req, res) {  
    ...  
  });  
  app.get('/shoping', function (req, res) {  
    ...  
  })  
}
```

Declaración del módulo /routes/rsongs.js

```
require("./routes/rsongs.js")(app, swig);
```

Incluir el módulo /routes/rsongs.js

Arquitectura y módulos

- Los módulos que declaran un **objeto** permiten acceder a sus variables y funciones
 - Dentro del propio objeto sus variables y funciones se referencian con **this**

```
module.exports = {  
  name : null,  
  lastname : null,  
  init : function(name) {  
    this.name = name;  
  },  
  greeting : function(personalized) {  
    if (personalized == true )  
      return "Hola " + this.name;  
    else  
      return "Hola";  
  }  
};
```

init nombre común
para el inicializador

this.name es la variable
nombre del objeto

Declaración del módulo /modules/person.js

```
var person = require("./modules/person.js");  
person.init("John");
```

Incluir el módulo /modules/person.js

Arquitectura y módulos

- La sintaxis de un objeto es muy distinta a la de una función
- Un **objeto** no es lo mismo que una **clase**

```
var objeto = {  
  <nombre_atributo> : <valor>,  
  <nombre_atributo> : <valor>,  
  <nombre_atributo> : <valor>,  
  <nombre_función> : function( <parámetros> ){  
  
    },  
  <nombre_función> : function( <parámetros> ){  
  
    }  
};
```

- **require(<path del módulo>)** *siempre retorna la misma instancia de objeto*
 - Aunque se incluyan varios **require** todos retornan la referencia al mismo objeto

Arquitectura y módulos

- Los módulos que declaran una **clase** permiten crear instancias
 - Dentro de la clase, sus variables y funciones se referencian con **this**

```
module.exports = class Person {  
  constructor(name) {  
    this.name = name;  
  }  
  greeting (personalized) {  
    if (personalized == true )  
      return "Hola " + this.name;  
    else  
      return "Hola";  
  }  
};
```

Clase Persona

Constructor

```
var Person = require("./modules/person.js");  
var person1 = new Person("John");  
var person2 = new Person("J");
```

2 instancias de
Persona

Arquitectura y módulos

- Las aplicaciones utilizan módulos:
 - **Externos** descargados normalmente con un gestor de paquetes (npm, yarn, etc) y contienen funcionalidad que puede ser común a muchas aplicaciones
 - body-parser, mongodb, twig, crypto, etc.
 - **Propios** implementación propia, normalmente específica o relativa a una aplicación
 - adsManager, routesAds, etc.
- Los módulos deben ser usados:
 - **Desde la aplicación Node-Express :**
 - Ejemplo: el módulo **body-parser** es usado por la aplicación en el procesamiento del body (parámetros POST)
 - **Desde otros módulos:**
 - Ejemplo: el módulo **routesAds** utiliza el módulo **adsManager** para acceder a los anuncios y **twig** para generar respuestas basadas en plantillas.

Arquitectura y módulos

- Alternativas de uso de módulos(externos o propios) desde otros módulos o partes de la app:
 - Caso 1: Obtener el objeto/función allí donde sea requerido (alternativa no muy mantenible, los cambios pueden ser costosos)

```
util = require("utils.js");
```

users.js

```
util = require("utils.js");
```

payments.js

- Caso 2: Obtener el objeto/función una vez y *enviarlo como parámetro* a otros módulos

```
util = require("utils.js");
```

app.js

```
module.exports = function(util)
```

```
module.exports = function(util)
```

- Caso: Obtener el objeto/función y *almacenarlo en variables de la app* (Será, accesibles desde cualquier parte, no conviene abusar de las variables de app)

```
app.set('util',require("utils.js"));
```

```
app.get('util');
```

```
app.get('util');
```

Arquitectura y módulos

- Usos de módulos integrados con la aplicación Express
 - Suelen ser módulos **vinculados a express**
 - Primero se obtiene el objeto/función correspondiente al módulo
 - En algunos casos se puede configurar
 - *Luego se integra en la app con **app.use(<objeto/funcion>)***
 - Otorga nuevas funcionalidades a la **app** (muchas veces transparentes, no se requiere referenciar al módulo específico para obtener la funcionalidad)
 - Ejemplo:
 - La app ya puede procesar cuerpos de peticiones (parámetros POST)

```
var bodyParser = require('body-parser');  
app.use(bodyParser.json());
```

- La app ya puede recibir ficheros en peticiones

```
var fileUpload = require('express-fileupload');  
app.use(fileUpload());
```

BASES DE DATOS MONGODB

Bases de datos MongoDB

- Son bases de datos **no relacionales (NoSQL)**
 - No existen tablas ni estructuras fijas que deban cumplir los datos almacenados
- *Orientadas a documentos, donde la información se almacena en formato BSON*
 - BSON es una versión ligera creada a partir de JSON
 - <https://www.mongodb.com/json-and-bson>
- Un **documento** contiene un **objeto BSON**, con atributos que pueden tomar diferentes valores (tipos simples, objetos, colecciones, etc.)

Bases de datos MongoDB

■ Ejemplo **documento**:

```
{
  name: "Cambiar ordenadores",
  computers: 3,
  attended: true,
  description: "Cambiar todos los ordenadores",
  details: {
    category: "mantenimiento",
    cost: 4233
  },
  incidents: [
    {
      description: "Inicio sin problemas",
      date: "23-06-2016"
    },
    {
      description: "Falta de material",
      date: "24-06-2016"
    }
  ]
}
```

Valor: tipo simple

Valor: objeto { }

Valor: colección []

Ventajas de MongoDB

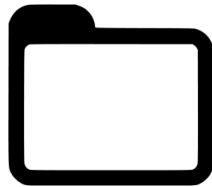
- Sin esquema previo (Schema Less)
 - Permite almacenar datos no estructurados
- Flexible
- Escalable y distribuida
- Alta disponibilidad
- Sintaxis sencillas para hacer consultas complejas
- Integración con lenguajes de programación populares
- Código abierto y bajo costo de propiedad

Desventajas de MongoDB

- No es adecuada para aplicaciones con transacciones
 - NO soporta transacciones ACID
 - Atomicity
 - Las transacciones son completas
 - Consistency -> Integridad
 - Cualquier transacción llevará a la base de datos desde un estado válido a otro también válido
 - Isolation -> Aislamiento
 - Esta propiedad asegura que una operación no puede afectar a otras.
 - Durability -> Persistencia
 - Esta propiedad asegura que una vez realizada la operación, esta persistirá y no se podrá deshacer aunque falle el sistema y que de esta forma los datos sobrevivan de alguna manera.
 - Soportado a partir de MongoDB 4.0 en algunas operaciones de escritura en un único documento
- Sin garantía de integridad de datos
- Menos soporte para consultas complejas.
 - No soporta relaciones entre entidades para consulta.
- Uso intensivo de recursos

Bases de datos MongoDB

- Los **documentos** se almacenan en **colecciones**
- Una **colección** es básicamente la carpeta donde se almacenan los documentos (agrupación)
- La **colección** no define la estructura de los documentos (no es una tabla)
 - Cada **documento** puede seguir una estructura diferente
 - La estructura de un documento puede ser modificada dinámicamente



Colección proveedores

```
{  
  "name" : "John",  
  "lastname" : "Doe",  
  "quality" : 10  
}
```

```
{  
  "center" : "uniovi",  
  "quality" : 10  
}
```

Bases de datos MongoDB

- Sobre las **colecciones** se realizan **operaciones** que permiten gestionar los documentos almacenados en la colección
 - **Colección.find**({ criterio de selección }): obtener documentos
 - **Colección.insertOne**({ documento }): insertar un nuevo documento
 - **Colección.updateOne**({criterio de selección } , { nuevo documento }): actualizar documentos
 - **Colección.deleteOne**({criterio de selección }): eliminar documento
 - Otros.

Bases de datos MongoDB

- Al guardar un documento, MongoDB agrega de forma automática un **_id : ObjectId**
- El **ObjectId** actúa como identificador único del documento
 - Se genera automáticamente
 - Compuesto por 12 Bytes
 - 4 bytes: Timestamp (marca de tiempo), momento de creación
 - 3 bytes: Identificador de la máquina (resumen hash)
 - 2 bytes: PID – identificador del proceso
 - 3 bytes: Contador incremental

```
> db.proyectos.find()  
{ "_id" : ObjectId("574449da40fb278c24332fa6"), "nombre"  
  "pcion" : "Cambiar todos los ordenadores" }
```

Servidor de bases de datos MongoDB

- **Instalación Local** descargando el servidor de la página oficial

<https://www.mongodb.com/es>

- Ejecutar el instalable y completar la instalación
- Crear la **carpeta** para almacenar las bases de datos, por ejemplo:
C:\data\db
- Acceder a la carpeta donde se instaló **\MongoDB\Server\3.4\bin** y ejecutar el comando de arranque del servidor:
mongod --dbpath C:\data\db

```
2016-10-16T18:00:04.543+0200 I - [initandlisten] Detected data files in C:\data\db\ crea
storage engine, so setting the active storage engine to 'wiredTiger'.
2016-10-16T18:00:04.544+0200 I STORAGE [initandlisten] wiredtiger_open config: create,cache_si
viction=(threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait
2016-10-16T18:00:04.716+0200 I NETWORK [HostnameCanonicalizationWorker] Starting hostname cand
2016-10-16T18:00:04.716+0200 I FTDC [initandlisten] Initializing full-time diagnostic data
t:/data/db/diagnostic.data'
2016-10-16T18:00:04.718+0200 I NETWORK [initandlisten] waiting for connections on port 27017
```

- La cadena de conexión será: **mongodb://localhost:27017/<nombre base de datos>** (Por defecto acceso libre sin usuario)
 - Sí la base de datos no existe, se crea al conectarse

Servidor de bases de datos MongoDB

- **Mongo en la Nube** usando un proveedor de cloud computing
 - **MongoDB Atlas** (y otros muchos proveedores) permiten la creación de bases de datos en la Nube
 - Permite servidores “elásticos” pudiendo cambiar entre servidores con más o menos recursos según el uso requerido
 - Ofrece **512mb** de almacenamiento de datos sin coste
 - Permite crear múltiples bases de datos
 - Por **seguridad** requiere la creación de un **usuario-password** para la base de datos
 - Obtenemos una cadena de conexión, por ejemplo:

`mongodb+srv://<dbuser>:<dbpassword>@<clustername>.xjf0khu.mongodb.net/
?retryWrites=true&w=majority`

Servidor de bases de datos MongoDB

- Desde <https://cloud.mongodb.com/> puede consultar todas las **colecciones** y **documentos** que se van creando.
 - Ejemplos colecciones: **canciones** y **usuarios**...

The screenshot shows the MongoDB Cloud interface for a database named 'tiendamusica'. The 'Collections' tab is selected and highlighted with a green box. Below the tabs, it shows 'DATABASES: 1' and 'COLLECTIONS: 4'. On the left, under the 'test' namespace, a list of collections is shown, with 'canciones' highlighted by a green box. On the right, the 'test.canciones' collection is selected, showing a size of 534B. Below this, there are tabs for 'Find' and 'Indexes', and a 'FILTER' button with a query string. At the bottom, it says 'QUERY RESULTS 1-6 OF 6'.

tiendamusica

Overview Real Time Metrics **Collections**

DATABASES: 1 COLLECTIONS: 4

+ Create Database

Q NAMESPACES

test

- canciones**
- comentarios
- compras
- usuarios

test.canciones

COLLECTION SIZE: 534B

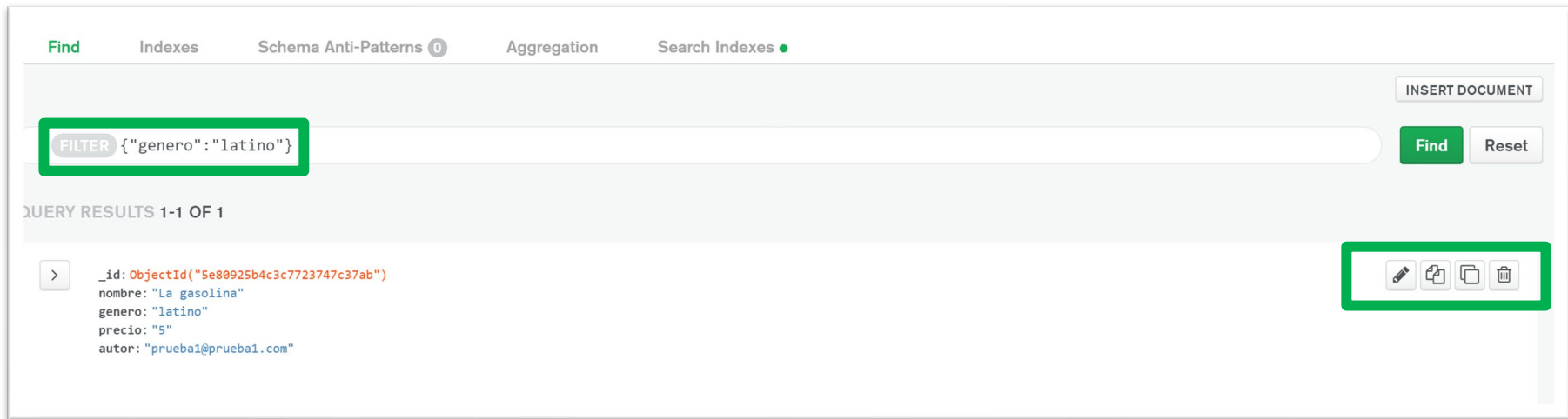
Find Indexes

FILTER {"filter": "e

QUERY RESULTS 1-6 OF 6

Servidor de bases de datos MongoDB

- Se puede ver el **contenido de los documentos**, realizar **búsquedas**, **insertar**, **borrar y modificar**.



Conexión a bases de datos MongoDB

- Para que una aplicación Node.js se conecte a una base de datos se requiere un **módulo** (librería)
- Cada motor de bases de datos utiliza un módulo propio
- El módulo **mongodb** es el driver oficial para MongoDB en Node.js
 - <https://mongodb.github.io/node-mongodb-native/>
 - No está incluido en el core de Node.js
- Actualmente se mantienen varias versiones release de **mongodb**: 3.X y 5.X cada una utiliza API muy diferente

4.7 Driver	Reference API
4.6 Driver	Reference API
4.5 Driver	Reference API
4.4 Driver	Reference API
4.3 Driver	Reference API
4.2 Driver	Reference API
4.1 Driver	Reference API
4.0 Driver	Reference API
3.7 Driver	Reference API
3.6 Driver	Reference API

RELEASE	DOCUMENTATION
Next Driver	Reference API
5.1 Driver	Reference API
5.0 Driver	Reference API
4.14 Driver	Reference API
4.13 Driver	Reference API
4.12 Driver	Reference API
4.11 Driver	Reference API
4.10 Driver	Reference API
4.9 Driver	Reference API
4.8 Driver	Reference API

Conexión a bases de datos MongoDB

- En la instalación del módulo se debe especificar la versión.

Especificar versión concreta

`npm install mongodb@4.1.4 --save`

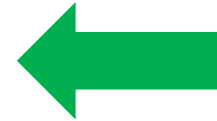
- Si no especificamos versión instala la que “considera la última”
- Con **require** añadimos el módulo **mongodb** a la aplicación

```
let express = require('express');  
let app = express();  
let mongo = require('mongodb');
```

Conexión a bases de datos MongoDB

- El módulo **mongodb** contiene todo lo necesario para conectarnos a la base de datos
 - Incluido el cliente **mongo.MongoClient**
 - El módulo **mongo** se envía a los módulos que realicen el acceso a datos.

```
require("./routes/rsong.js")(app, swig, mongo);
```



- Para conectarse a la base de datos podemos usar una **cadena de conexión**:
 - `mongodb+srv://<dbuser>:<dbpassword>@<clustername>.xjf0khu.mongodb.net/?retryWrites=true&w=majority`
 - Es recomendable guardar la URL en las **variables de la aplicación**
 - Así se podrá usar en todos los módulos de la aplicación

```
app.set('connectionStrings', 'mongodb+srv://<dbuser>:<dbpassword>@<clustername>.xjf0khu.mongodb.net/?retryWrites=true&w=majority');
```

Conexión a bases de datos MongoDB

- **mongo.MongoClient** permite **conectarse** a una base de datos Mongo
 - La función **connect** requiere los parámetros:
 - URL de conexión
 - Función *manejadora (Handler)*, con dos parámetros:
 - **err** -> en caso de haber errores este parámetro toma valor, incluye el mensaje del error
 - **dbClient**: -> referencia al cliente de la base de datos, sobre este objeto se realizan las acciones (insertar, consultar, borrar, etc.)
 - **dbClient.db("MyDatabase")** -> devuelve la conexión a una base de datos

```
mongo.MongoClient.connect(app.get('connectionStrings'), function(err, dbClient)
{
    if (err) {
        // Error al conectar
    } else {
        // usar "dbClient" para realizar acciones (insertar, etc.)
        const database = dbClient.db("MyDatabase");
    }
});
```

Gestión de datos con MongoDB

- Usaremos el objeto **dbClient.db("MyDatabase")** para gestionar los datos
 - Ejemplo: const **database** = dbClient.db("MyDatabase");
 - **database.collection(<nombre colección>)** da acceso a una colección
 - Se pueden referenciar incluso colecciones no existentes
 - Sí guardamos un documento en una colección no existente se creará la colección.
- Sobre la colección se realizan las acciones, por ejemplo:
 - **insertOne(objeto JSON, función manejadora(err, resultado))** -> para guardar un nuevo documento
 - El **resultado** de la función manejadora depende de la **acción**:
 - Las inserciones retornan el documento insertado (con su **_id**)
 - Las búsquedas retornan **listas** de documentos
 - Etc.
- Todas las acciones (al igual que la conexión) son **asíncronas**
 - Cuando la acción se completa se invoca la función manejadora
 - Podemos usar **callback, promesas y async/await**

Gestión de datos con MongoDB

- Ejemplo **insert** usando callback (descontinuado)

```
var song = {
  name : req.body.name,
  gender : req.body.gender
}

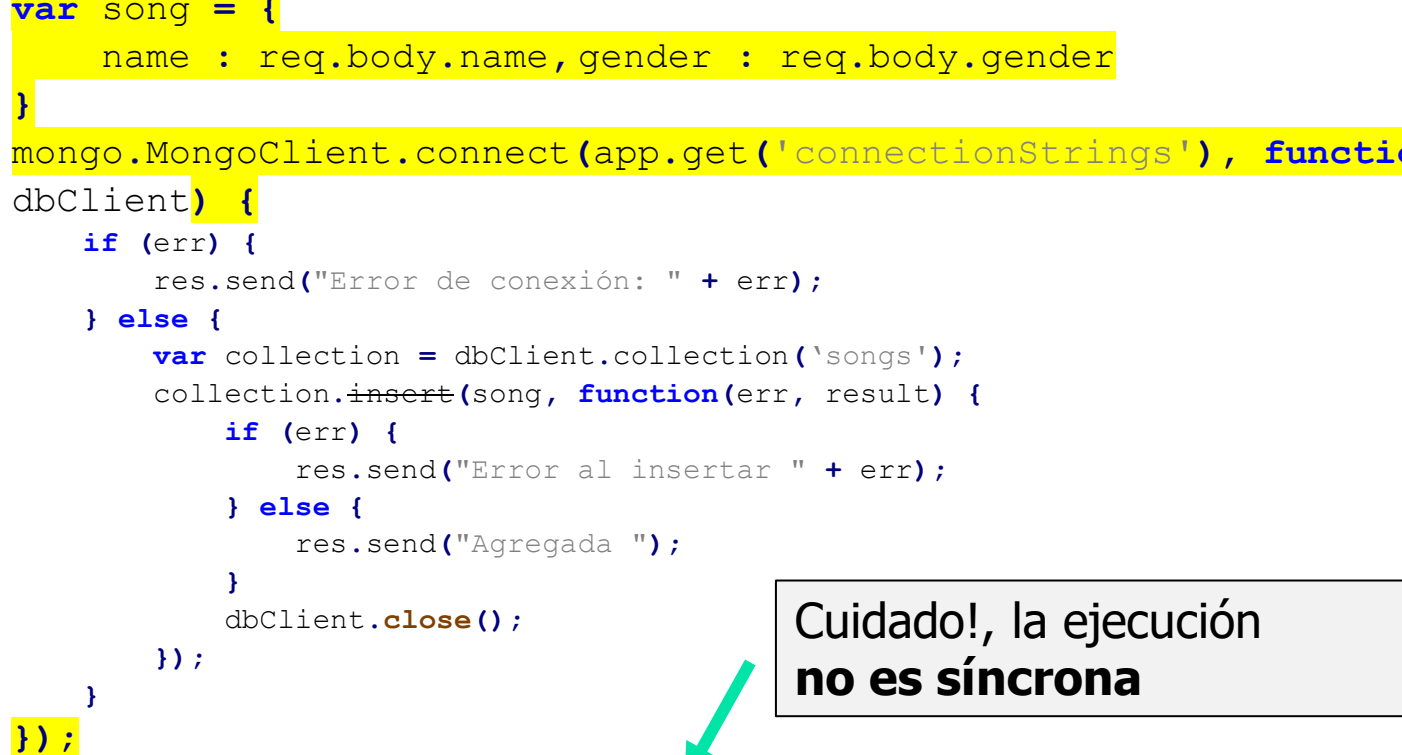
mongo.MongoClient.connect(app.get('connectionStrings'), function(err, dbClient) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = dbClient.collection('songs');
    collection.insert(song, function(err, result) {
      if (err) {
        res.send("Error al insertar " + err);
      } else {
        res.send("Agregada ");
      }
      dbClient.close();
    });
  }
});
```

manejador

Una vez acabado recomendado **cerrar la conexión**

Gestión de datos con MongoDB

- Debemos tener muy claro el concepto de ejecución **asíncrona**
- El código se va ejecutando por **fases** usando las funciones **manejadoras**



```
var song = {
  name : req.body.name, gender : req.body.gender
}

mongo.MongoClient.connect(app.get('connectionStrings'), function(err,
dbClient) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = dbClient.collection('songs');
    collection.insert(song, function(err, result) {
      if (err) {
        res.send("Error al insertar " + err);
      } else {
        res.send("Agregada ");
      }
    });
    dbClient.close();
  }
});
});
```

Cuidado!, la ejecución
no es síncrona

Sí respondemos aquí res.send(. . .) No se ejecuta la conexión a Mongo
Más código

Gestión de datos con MongoDB

- Debemos tener muy claro el concepto de ejecución **asíncrona**

2º Fase connect



```
var song = {
  name : req.body.name,
  gender : req.body.gender
}

mongo.MongoClient.connect(app.get('connectionStrings'), function(err, dbClient) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = dbClient.collection('songs');
    collection.insert(song, function(err, result) {
      if (err) {
        res.send("Error al insertar " + err);
      } else {
        res.send("Agregada ");
      }
      dbClient.close();
    });
  }
});

Más código
Más código
```

Gestión de datos con MongoDB

- Debemos tener muy claro el concepto de ejecución **asíncrona**

```
var song = {
  name : req.body.name,
  gender : req.body.gender
}
mongo.MongoClient.connect(app.get('connectionStrings'), function(err, dbClient) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = dbClient.collection('songs');
    collection.insert(song, function(err, result) {
      if (err) {
        res.send("Error al insertar " + err);
      } else {
        res.send("Agregada ");
      }
      dbClient.close();
    });
  }
});
```

Más código
Más código

3º Fase insert

Respuesta final
En caso de Éxito

Gestión de datos con MongoDB

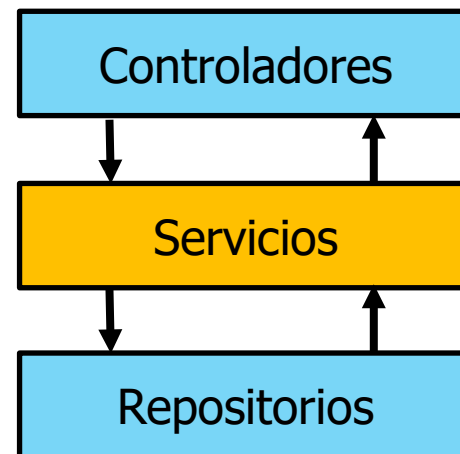
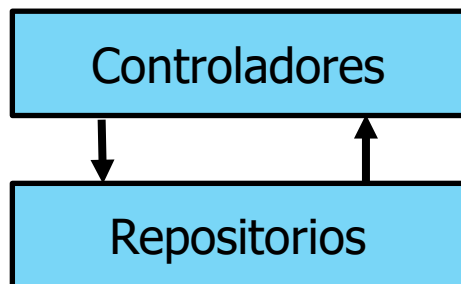
■ Ejemplo **insertOne** usando promesas

```
...
insertSong: function (song, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('connectionStrings'),
function (err, dbClient) {
    if (err) {
        funcionCallback(null);
    } else {
        const database = dbClient.db("musicStore");
        let collection = database.collection('songs');
        //Con promesa
        collection.insertOne(song)
            .then(result => funcionCallback(result.insertedId))
            .catch(err => funcionCallback({error: err.message}));
        dbClient.close();
    }
    });
},
...

```

Arquitectura: acceso a datos

- Debemos encapsular el acceso a datos en **uno o varios módulos**
 - Dependiendo del número de **entidades** y **operaciones**, hay que valorar:
 - Un único módulo para varias entidades relacionadas
 - Un módulo para cada entidad
- Para **lógicas de negocio simples** los controladores podrían utilizar los **módulos de acceso a datos**
 - Por ejemplo: sí la aplicación solo realiza operaciones CRUD básicas
- Para lógica compleja implementaríamos **módulos de servicios**



Arquitectura: acceso a datos

- Podemos definir un **módulo** como **objeto** que encapsule las operaciones
- Como las operaciones son **asíncronas** deben recibir una **función de callback** (retorno), **No usamos return**
 - **Función de callback:** se invoca al finalizar la operación asíncrona, por ejemplo: para enviarle el **id** del objeto insertado.

```
...
insertSong: function (song, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('connectionStrings'),
    function (err, dbClient) {
        if (err) {
            funcionCallback(null);
        } else {
            const database = dbClient.db("musicStore");
            let collection = database.collection('songs');
            //Con promesa
            collection.insertOne(song)
                .then(result => funcionCallback(result.insertedId))
                .catch(err => funcionCallback(null))
            dbClient.close();
        }
    });
},
```

Retorno de Respuesta

Retorno Error

Arquitectura: acceso a datos

- Por ejemplo: uso de un módulo de acceso a datos desde un controlador
 - **songsRepository** es la referencia al módulo

```
app.post("/song", function(req, res)
  var song = {
    name : req.body.name,
    genero : req.body.genero
  }
```

Función de callback

Al acabar de **insertar**

Debe recibir la **id** de la canción insertada

```
funcionCallback(result.insertedId);
```

```
songsRepository.insertSong(song, function(id) {
```

```
  if (id == null) {
    res.send("Error al insertar ");
  } else {
    res.send("Agregada id: "+ id);
  }
}
```

```
});
```

```
});
```



OPERACIONES CRUD

Insertar un documento

- **Método InsertOne()** - Inserta un documento en una colección
 - Si los documentos no contienen el campo **_id** lo agrega automáticamente
 - Si la colección especificada no existe, el método insertOne() crea la colección.
 - Parámetros de **insertOne()**:
 - Documento que contiene los campos y valores que desea almacenar.
 - *options (opcional)
 - **writeConcern** (opcional). Un documento que expresa la inquietud de escritura.
 - Omitir el uso de la writeConcern de escritura predeterminada.
 - **Ver más:** <https://mongodb.github.io/node-mongodb-native/4.4/interfaces/InsertOneOptions.html>
 - También puede pasar una función de callback como un tercer parámetro opcional
 - Retorna un documento (objeto) conteniendo:
 - Un campo *insertedId* con el valor **_id** del documento insertado.
 - Un booleano *acknowledged* indica si la operación de inserción fue exitosa o no (Si la operación se ejecutó con writeConcern).
 - <https://docs.mongodb.com/v4.0/reference/method/db.collection.insertOne>

Insertar un documento

■ Ejemplo insertar usando promesas

```
...
insertSong: function (song, functionCallback) {
    this.mongo.MongoClient.connect(this.app.get('connectionStrings'), function
(err, dbClient) {
        if (err) {
            functionCallback(null);
        } else {
            const database = dbClient.db("musicStore");
            let collection = database.collection('songs');
            //Con promesa
            collection.insertOne(song)
                .then(result => functionCallback(result.insertedId))
                .catch(err => functionCallback({error: err.message}));
            dbClient.close();
        }
    });
},
```

Colección

Documento a insertar

insertedId -> _id del documento insertado

Insertar varios documento

- **Método InsertMany()** - Inserta un array de documentos en una colección
 - Si los documentos no contienen el campo **_id** lo agrega automáticamente
 - Si la colección especificada no existe, el método insertMany() crea la colección.
 - Parámetros de **insertMany()**:
 - Array de documentos que se quieren almacenar
 - *options (opcionales).
 - **writeConcern** (opcional). Un documento que expresa la inquietud de escritura.
 - Omitir el uso de la writeConcern de escritura predeterminada.
 - **Ordered** (opcional): si es true, esta opción evita que se inserten documentos adicionales si uno falla.
 - **Ver más:** <https://mongodb.github.io/node-mongodb-native/4.4/interfaces/InsertOneOptions.html>
 - Retorna un documento (objeto) conteniendo:
 - Un campo **insertedIds**: lista de _id de los documentos insertados.
 - Un booleano **acknowledged** indica si la operación de inserción fue exitosa o no (Si la operación se ejecutó con writeConcern).
 - **insertedCount**: El número de documentos insertados en la operación.
 - <https://docs.mongodb.com/drivers/node/current/usage-examples/insertMany/>

Insertar varios documentos

■ Ejemplo insertar usando Async/Await

```
...
Async function (song, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('connectionStrings'),
function (err, db) {
    if (err) {
        funcionCallback(null);
    } else {
        const database = dbClient.db("musicStore");
        const collection = database.collection('songs');
        //Con await
        const options = { ordered: true };
        const result = await collection.insertMany(songList, options)
        funcionCallback(result.insertedIds)
        db.close();
    }
});
},
```

Colección

Documentos a insertar

insertedIds -> lista de _id de los documentos insertados

Borrar un documento

- **Método DeleteOne()** - Elimina un documento de una colección
 - Utiliza una consulta para filtrar los documentos.
 - Se elimina el primer documento que coincida con la consulta.
 - Parámetros de **DeleteOne()**:
 - **filter** (filtro): selector para la operación de eliminar
 - {"type" : "casa"} = los documentos de tipo casa
 - {"price": { \$gte: 31 }} = documentos con precio mayor o igual que 31
 - \$gt . greater than. Mayor que
 - \$gte – greater than or equal . Mayor o igual que
 - \$lt – less than, Menor que
 - \$lte – less than or equal. Menor o igual que
 - {\$or : [{"age" : 20}, {"age" : 30}, {"age": 40}]} = documentos con edad 20, 30 o 40 . **OR**
 - {\$and : [{"age" : 40}, {"empresa" : "CSC"}]} = documentos con edad 40 y empresa CSC . **AND**
 - options (opcionales):
 - **writeConcern**. Un documento que expresa la inquietud de escritura.
 - Omitir el uso de la writeConcern de escritura predeterminada.
 - **Ver más:** <https://mongodb.github.io/node-mongodb-native/4.4/interfaces/DeleteOptions.html>
 - También puede pasar un método de callback como un tercer parámetro opcional
 - Retorna un documento (objeto) conteniendo:
 - Un booleano **acknowledged** indica si la operación de borrado fue exitosa o no (Si la operación se ejecutó con writeConcern).
 - **deletedCount**: El número de documentos borrados (debe ser 1).
- **Método DeleteMany()**: Elimina múltiples documentos de una colección
 - <https://docs.mongodb.com/drivers/node/current/usage-examples/deleteMany/>

Borrar un documento

■ Ejemplo deleteOne()

```
// Con Await / declarar la funcion como async
const query = {name: "James" };
const database = dbClient.db("musicStore");
const collection = database.collection('songs');
const result = await collection.deleteOne(query);
if (result.deletedCount === 1) {
    callbackControlador(result.deletedCount);
} else {
    callbackControlador(null);
}
db.close();

// Con promesas
const query = {name: "James" };
const database = dbClient.db("musicStore");
const collection = database.collection('songs');
collection.deleteOne(query)
    .then(result => callbackControlador(result.deletedCount))
    .catch(err => callbackControlador(null));
db.close();
};
```

Número de documentos afectados
por la acción **deleteOne**



Actualizar un documento

- Método **updateOne()** - Actualiza un documento de una colección
 - Acepta un documento de filtro y un documento de actualización.
 - Parámetros de updateOne():
 - **Filter (Filtro)**: selector para el documento a modificar.
 - **Documento**: nuevo documento que sustituye a los seleccionados por el criterio
 - *Options (opcionales).
 - **Upsert**: establecer la opción upsert a true para crear un nuevo documento si ningún documento coincide con el filtro.
 - **writeConcern**. Un documento que expresa la inquietud de escritura.
 - **Ver más**: <https://mongodb.github.io/node-mongodb-native/4.4/classes/Collection.html#updateOne>
 - Retorna un documento (objeto) conteniendo:
 - **acknowledged**: indica si la operación de inserción fue exitosa o no (Si la operación se ejecutó con writeConcern).
 - **matchedCount**: número de documentos que coincidieron con el filtro.
 - **modifiedCount**: número de documentos que fueron modificados.
 - **upsertedId**: identificador del documento insertado si se produjo una inserción (upser).
 - Método **UpdateMany()**: actualiza múltiples documentos de una colección
 - <https://docs.mongodb.com/drivers/node/current/usage-examples/updateMany/>

Actualizar un documento

■ Ejemplo de actualización

- Sustituye **completamente** el documento que cumple con el criterio de selección por el nuevo documento

```
const filter = { name : "James"};
const newPerson = { name : "R", lastName : "R"};
const options = { upsert: true } //crear un documento si no hay
documentos que coincidan con el filtro
const collection = db.collection('songs');
const result = await collection.updateOne(filter, newPerson,
options);
```

```
if (result.modifiedCount <= 0) {
  callbackController(null);
} else {
  callbackController(result);
}

db.close();
});
```

Pre-update

```
{
  "name" : "James",
  "lastName" : "J",
  "country" : "es",
  "language" : "es"
}
```

Post-update

```
{
  "name" : "R",
  "lastName" : "R"
}
```

Actualizar un documento

- Ejemplo actualización con **{ \$set:** objeto con atributos **}**
 - **Sustituye o agrega** los nuevos atributos al documento

```
var filter = { name : "J"};
var attributes = { lastName: "R", age : 40};
const options = { upsert: true }
const collection = db.collection('songs');
const result = await collection.updateOne(filter, { $set: attributes }, options);
if (result.modifiedCount <= 0) {
    callbackController(null);
} else {
    callbackController(result);
}
db.close();
});
```

Pre-update

```
{
  "name" : "J",
  "lastName" : "J",
  "country" : "es",
  "language" : "es"
}
```

Post-update

```
{
  "name" : "J",
  "lastName" : "R",
  "age" : 40,
  "country" : "es",
  "language" : "es"
}
```

Buscar documentos

- Metodo **FindOne()**: Realiza una búsqueda de un documento por criterio
 - Utiliza un documento de consulta para filtrar los documentos
 - Si no proporciona un documento de consulta, devuelve todos los documentos de la colección
 - El criterio selector, se expresa en formato JSON, por ejemplo:
 - {} = todos los documentos
 - {"type" : "casa"} = los documentos de tipo casa
 - {"type" : "casa", "metros" : 100} = los documentos de tipo casa y metros 100
 - Es equivalente a utilizar un AND
 - {"price":{ \$gte: 31 }}= documentos con precio mayor o igual que 31
 - \$gt . greater than. Mayor que
 - \$gte – greater tan or equal . Mayor o igual que
 - \$in – valor contenido en un array
 - \$nin – valor NO contenido en un array
 - \$lt – less than, Menor que
 - \$lte – less than or equal. Menor o igual que
 - {\$or : [{"age" : 20}, {"age" : 30}, {"age":40}]} = documentos con edad 20, 30 o 40 . **OR**
 - {\$and : [{"age" : 40}, {"empresa" : "CSC"}]} = documentos con edad 40 y empresa CSC . **AND**
 - Se puede definir opciones de consulta adicionales para configurar el documentos, como:
 - **Sort**: ordenar por un criterio
 - **Projection**: Incluir solo los campos especificado en el documento devuelto
- Método **Find()**: consultar varios documentos en una colección

Buscar documentos

- Sobre el **find()** se aplica (1-N) una operaciones para obtener los resultados, por ejemplo:
 - **toArray (callback (err, resultado))** -> El **resultado** es un array de documentos

```
const query = { title: "despacito" };
const options = { sort: { "title": 1 }, projection: { _id: 0,
title : 1, author : 1}};
const collection = db.collection('songs');
const cursor = collection.find(query, options);
const songs = await cursor.toArray();
if (songs.length <= 0) {
    callbackControlador(null);
} else {
    callbackControlador(songs);
}
db.close();
};
```

Buscar documentos

- Antes de obtener los documentos podemos aplicar **filtros**, por ejemplo:
 - **skip (número)** : saltarse los n primeros registros
 - **limit (número)** : limitar el número de registros
 - Usaremos estas funciones para implementar paginación
- Ejemplo para obtener 3 documentos

```
...  
const sort = { length: -1 };  
const limit = 3;  
const collection = db.collection('songs');  
const cursor = collection.find({}).sort(sort).limit(limit);  
const songs = await cursor.toArray();  
if (songs.length <= 0) {  
    callbackControlador(null);  
} else {  
    callbackControlador(songs);  
}
```

Filtra el resultado antes de
obtener el array



ObjectID

- La transformación de objeto Mongo a JavaScript es **automática**

```
collection.find({}).toArray(function(err, usuarios) {  
    funcionCallback(usuarios);  
    db.close();  
});
```

Array de objetos. Cada objeto tiene los datos de **usuario**


- Los objetos JS recuperados de mongo tienen un **`_id` : ObjectId**
 - **ObjectID** por defecto no es un tipo simple
 - El valor de este atributo es una **instancia de ObjectId**
 - Para acceder al valor como cadena: `<objeto>._id.toString()`
 - Ejemplo en JavaScript:

```
var usr = usuarios[0];  
var a = usr._id; // ObjectId  
var b = usr._id.toString(); // String
```

ObjectID

- Considerar ObjectID en los criterios de selección por **_Id**
 - Tipo ObjectID no es Tipo String

```
app.get('/usuario/:id', function (req, res) {  
    var criterio = { "_id" : req.params.id };
```



Los **_id** NO son de tipo String

- Posible solución: convertir el String recibido a ObjectId
 - El módulo **mongodb** permite crear ObjectIds, con la función: **mongo.ObjectId(String)**.

```
app.get('/usuario/:id', function (req, res) {  
    var objectID =  
    usersRepository.mongo.ObjectId(req.params.id);  
    var criterio = { "_id" : objectID };
```

ENCRIPCIÓN (CIFRADO) Y AUTENTICACIÓN Y AUTORIZACIÓN

Encriptación (Cifrado)

- El módulo **crypto** permite cifrar (encriptar) y descifrar (desencriptar)
 - <https://nodejs.org/api/crypto.html>
- Está incluido en el Core de Node.js (No hay que instalarlo)
 - El objeto **crypto** se obtiene con un **require**

```
var crypto = require('crypto');
```
- Es necesario cifrar las contraseñas y cualquier otra información que sea sensible
- Permite múltiples algoritmos de cifrado:
 - sha256, sha512, otros.
- Permite realiza múltiples codificaciones :
 - hex, latin1, base64, etc.

Encriptación (Cifrado)

- Requiere definir una "clave de cifrado" o "secreto"
 - **createHmac(<tipo>, <secreto>):** crea un objeto para realizar el cifrado

```
secreto = 'abcdefg';  
valor = "342434";  
encryptor = crypto.createHmac('sha256', secreto);
```

- **update(<valor a cifrar>):** retorna el valor cifrado
 - **digest(<tipo>):** especifica la codificación del valor cifrado

```
encryptedValue = encryptor.update(valor).digest('hex');
```

Autenticación y autorización

- La autenticación consiste en **validar la identidad** de un usuario
- Como mínimo los usuarios se identifican usando:
 - Username : identificador único, ID, DNI, nombre, email, etc.
 - Password: contraseña del usuario
- Muchos frameworks proveen sistemas de autenticación/autorización
 - Estos sistemas siguen sus propios enfoques (diferentes entre ellos)
 - Son de muy alto nivel, suelen “abstraer” los conceptos
 - Ejemplo: Spring Security en Spring, Express-authentication en Express, etc.
- Implementar un **sistema propio** la alternativa a usar los provistos por los frameworks

Autenticación


- Un proceso de implementación de Autenticación podría ser:
 1. Definir un controlador que reciba la petición POST con los parámetros
 - username (en este caso email) y password

```
app.post("/login", function(req, res) {  
    var email = req.body.email;  
    var password = req.body.password;
```


2. Realizar una búsqueda en los usuarios por ambos criterios
 - En la base de datos el password está encriptado

```
var seguro = encriptador.update(password).digest('hex');  
var criterio = {  
    email : email,  
    password : seguro  
}  
userRepository getUsers(criterio, function(users) {  
  
});
```

Password encriptado



Array de usuarios que
cumplen el criterio



Autenticación

- Un proceso de implementación de Autenticación podría ser:
 3. ¿Retorna algún usuario con ese criterio de búsqueda?
 - Null o 0 – **No se ha autenticado**, redireccionar a la URL apropiada
 - 1 usuario – **Se ha autenticado**, redireccionar a la URL apropiada

```
userRepository getUsers(criterio, function(users) {  
    if (users == null || users.length == 0) {  
        // No se ha autenticado  
    } else {  
        // Se ha autenticado  
    }  
});
```

Sesión y Autorización

- Una vez el usuario se autentica con éxito debemos **recordarlo**
 - El usuario con **email = J** está autenticado en el navegador X
- El objeto **sesión** es clave para identificar navegadores/clientes autenticados
- La sesión de express es un módulo externo **express-session**
 - <https://github.com/expressjs/session>


```
npm install express-session --save
```

- La función sesión se obtiene con **require**
 - La función puede recibir muchísimos parámetros de configuración opcionales. Algunos de los más comunes son:
 - **secret**: cadena de texto que se usará para cifrar la sesión
 - **resave**: (**true** / false) guarda la sesión en el almacén en cada petición, incluso aunque no haya sido modificada durante la petición
*Dependiendo del almacén de sesiones es necesario activarlo
 - **saveUninitialized**: (**true** / false) no esta inicializada hasta que no se modifica.

Sesión y Autorización

- La función sesión se integra con la aplicación con **app.use()**
- Ejemplo de configuración de sesión:

```
var app = express();  
  
var expressSession = require('express-session');  
app.use(expressSession({  
  secret: 'abcdefg',  
  resave: true,  
  saveUninitialized: true  
}));
```



La configuración de express-sesión se envía en un **objeto**
El objeto define: **secret, resave y saveUninitialized**

Sesión y Autorización

- La sesión es accesible desde todas las peticiones (request)
- Sus atributos se pueden leer/escribir mediante:
req.session.<clave del atributo>
- El usuario autenticado correctamente se almacenará en la **sesión**
 - Se debe guardar un valor que le identifique de forma única, ejemplo : **email**

```
userRepository.getUser(criterio, function(users) {  
    if (users == null || users.length == 0) {  
        req.session.user = null;  
        // respuesta no autenticado  
    } else {  
        req.session.user = users[0].email;  
        // respuesta autenticado  
    }  
});
```


Sesión y Autorización

- Para eliminar de sesión un usuario (desautenticar) podemos optar por:
 - Destruir la sesión **req.session.destroy()**
 - Poner a **null** el atributo que identifica al usuario

```
app.get('/logout', function (req, res) {  
  req.session.user = null;  
  res.send("Usuario desconectado");  
})
```

Sesión y Autorización

- La **autorización** debe comprobar si el cliente tiene permiso para acceder a las URLs de la aplicación
- La aplicación puede consultar en todo momento si hay un usuario autenticado utilizando la **sesión**.
- Ejemplo:

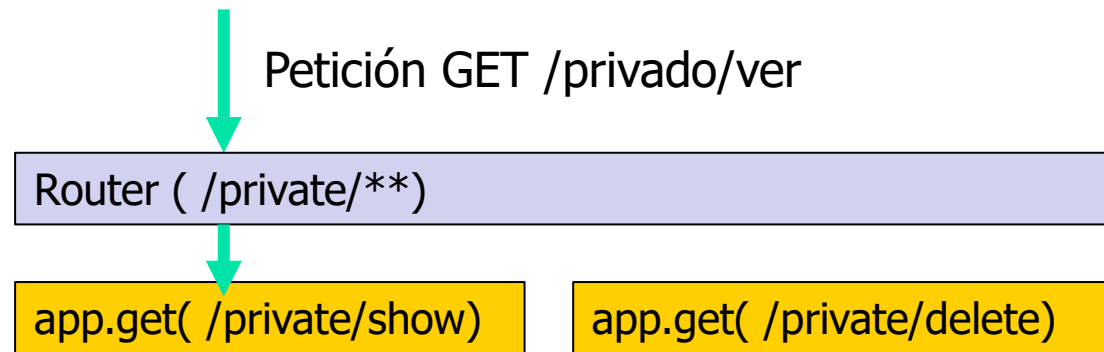
```
app.get('/private', function (req, res) {  
  if ( req.session.user == null){  
    // NO autenticado!  
    res.redirect("/login");  
    return;  
  }  
})
```

Si **usuario == null** no hay
usuario autenticado
No puede entrar en **/privado**

- No es nada apropiado realizar el control de autorización en las funciones app.get/post
 - Mala arquitectura, replicación de código, dificultad para realizar modificaciones
 - Solo sirve para controlar URLs declaradas en app no directorios (como /public, etc.)

Enrutadores

- Los **enrutadores** permiten definir funciones que procesan peticiones
 - Procesar una petición de forma similar a un **app.get**
- Si declaramos el uso de **enrutador** antes de agregar las URLs **app.get/post** procesará las peticiones antes que ellas



- La función del enrutador puede:
 - Ejecutar cualquier lógica de negocio
 - Dejar correr la petición (para que la procese el siguiente elemento)
 - Cortar la petición (por ejemplo: redireccionándola)

Enrutadores

- Un enrutador se crea con **express.Router()**
- Con **.use(<func>)** se le agrega una función manejadora
 - La función es similar la utilizada app.get() pero con un parámetro adicional **next**.
 - **next** es una función que deja correr la petición
- Ejemplo de creación de un enrutador

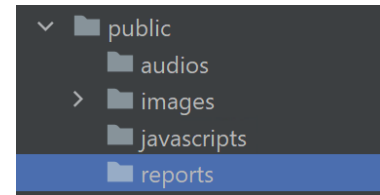
```
var routerAuthentication = express.Router();
routerAuthentication.use(function(req, res, next) {
  if ( req.session.user )
    // Hay usuario autenticado
    next();
  else
    // No hay usuario autenticado
    res.redirect("/login");
});
```

Si hay usuario autenticado
deja correr la petición

Si no hay usuario autenticado
redirecciona a /login

Enrutadores

- Una vez creado el enrutador se **agrega a la aplicación**
app.use(<ruta donde se aplica>, enrutador)
- Un mismo enrutador se puede aplicar en muchas rutas
- Por ejemplo:



```
app.use("/private/", routerAuthentication);  
app.use("/reports/", routerAuthentication);
```

```
app.use(express.static('public'));
```

```
app.get("/private/show", function(req, res)  
app.get("/private/delete", function(req, res)  
...
```

Agregar el enrutador a la aplicación en
/privado/
/informes/
**A todas las peticiones
GET , POST, ETC.**

Enrutadores

- En orden en que se agregan los enrutadores, directorios y respuestas (get/set) es crítico
- El orden determina quien responderá a la petición
- La función **next()** de los routers deja continuar la petición

GET /private/show

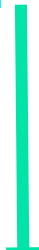


① `app.use("/private/", routerAuthentication);`
`app.use("/reports/", routerAuthentication);`

`app.use(express.static('public'));`

Solo si el router ejecuta
next() pasará al siguiente

② `app.get("/private/show", function(req, res)`
`app.get("/private/delete", function(req, res)`
`...`



Enrutadores

- Sí el orden no es correcto a la petición será respondida por quien no pretendíamos
- MAL -> Ejemplo 1: petición get



GET /private/show

```
app.use(express.static('public'));
```

① `app.get("/private/show", function(req, res)`
`app.get("/private/delete", function(req, res)`

Responde a la petición
se termina el ciclo

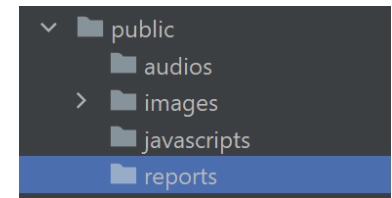
```
app.use("/private/", routerAuthentication);  
app.use("/reports/", routerAuthentication);
```

Nunca se ejecuta

Enrutadores

- Sí el orden no es correcto a la petición será respondida por quien no pretendíamos
- MAL -> Ejemplo 2: acceso a un recurso

GET /reports/1.pdf



① `app.use(express.static('public'));`

```
app.get("/private/show", function(req, res)
app.get("/private/delete", function(req, res)
```

Responde a la petición
se termina el ciclo

```
app.use("/private/", routerAuthentication);
app.use("/reports/", routerAuthentication);
```

Nunca se ejecuta

Enrutadores

- Se pueden **aplicar múltiples enrutadores** sobre un mismo path
- Se delega una única acción en cada enrutador
- Por ejemplo:
 - routerAutentication -> comprueba si hay usuario en sesión
 - routerBaneoIPs -> comprueba si la IP está en una lista negra

GET /private/show

① `app.use("/", routerBaneoIps);`

② `app.use("/private/", routerAutentication);`
`app.use("/reports/", routerAutentication);`

`app.use(express.static('public'));`

③ `app.get("/private/show", function(req, res)`

Solo si el router ejecuta **next()** pasará al siguiente

Solo si el router ejecuta **next()** pasará al siguiente

Enrutadores

- Una buena recomendación de diseño es agrupar las URLs por **su nivel de autorización** de forma jerárquica
- Ejemplo de una aplicación que gestiona anuncios donde:
 - **Todos los usuarios** pueden **ver** y **reservar** anuncios
 - **Solo los propietarios del anuncio** pueden **modificar** y **eliminar**
- Podríamos usar las siguientes URLs:
 - `/user/ads/show/:id`
 - `/user /ads/reserve/:id`
 - `/user/owner/ads/update/:id`
 - `/user/owner/ads/delete/:id`
- Donde habría dos enrutadores
 - `routerUser` -> comprueba que hay un usuario en sesión
 - `routerOwner` -> comprueba que es el propietario del anuncio



Subida de ficheros



Subida de ficheros

- Se requiere el modulo externo **express-fileupload**

```
npm install express-fileupload --save
```

- Se obtiene la función **express-fileupload** con **require**

```
var fileUpload = require('express-fileupload');
```

- Se agrega el objeto a la aplicación express (app.use())
 - La subida de ficheros ya estará disponible en la aplicación

```
app.use(fileUpload());
```

Subida de ficheros

- La petición (req) puede contener ficheros
 - Se accede a ellos con **req.files.<clave>**
 - Por ejemplo:
 - En un formulario que incluye el input de tipo **file** con clave **foto**
 - Debemos recordar el **enctype** de tipo **multipart/form-data**

```
<form method="POST" action="/savephoto" enctype="multipart/form-data">  
  <input type="file" name="photo" accept=".jpg" />  
  <input type="submit" value="Enviar" />  
</form>
```

- Como se procesa el fichero:
 - Se almacena en una **variable**
 - Se **copia en un directorio**
 - Elegimos el **directorio** y **nombre** del fichero
 - Podemos usar la función **file.mv(<directorio>,callback())**

Subida de ficheros

- Ejemplo de **file.mv(<directorio>,callback())**

Asegurarse de que no es **null**

```
if (req.files.photo != null) {  
    var photo = req.files.photo;  
    photo.mv('public/photos/' + id + '.jpg', function(err) {  
        if (err) {  
            // ERROR  
        } else {  
            // EXITO  
        }  
    });  
}
```

Solo si hay error la variable **err** tendrá un valor

Subida de ficheros

- El **path** donde se guarda el fichero es importante por ejemplo:
 - Directorio de acceso web **public/*** si queremos incluir la foto en la web
 - Directorios privados sí queremos que la petición pase por un controlador
- El **nombre** con el que se salva el fichero suele ser especificado por la lógica de negocio
 - Evita conflictos de nombres
 - Nombres o rutas que permitan asociar el fichero a un usuario asociar (por ejemplo: ID del usuario)
- En algunos casos es necesario hacer **comprobaciones de autorización** para acceder a ficheros de **directorio de acceso web**
 - **/static/reports** solo pueden acceder usuarios registrados.
 - **/static/photos/31** solo puede acceder el usuario 31



Manejo de errores



Captura de errores

- Por **defecto** y en **fase de desarrollo** se suele dejar que la aplicación propague errores
- La traza de error ofrece información útil para el desarrollador
- Por ejemplo: solicitud con un id mal formado no es ObjectId()
 - <http://localhost:8081/ads/RRRRR>
 - Al intentar formar un ObjectId(RRRR) produce un error

```
Error: Argument passed in must be a single String of 12 bytes or a string of 24 hex characters
    at new ObjectId (C:\Dev\workspaces\NodeApps\sdi-lab-node\node_modules\bson\lib\bson\objectid.js:57:11)
    at Function.ObjectId (C:\Dev\workspaces\NodeApps\sdi-lab-node\node_modules\bson\lib\bson\objectid.js:38:43)
```

- No debemos mostrar nunca esta información en producción
 - No es descriptiva para los usuarios
 - Potencialmente peligrosa, pueden detectar versiones de las tecnologías que utilizamos y buscar vulnerabilidades
 - Top 6 vulnerabilidad web OWASP (2007) – Filtrado de información y manejo inapropiado de errores. Actualmente no figura en el Top 10

Captura de errores

- Existen varios mecanismos para capturar los **errores en última instancia**
 - Cada función debería controlar todos sus errores, pero lograrlo con todos los posibles errores puede ser muy complejo
- Una forma global de capturar errores es incluir una función en la **aplicación que capture los errores controlados**
 - La incluimos con **app.use(función)** como elemento final de la aplicación
 - Sí detecta un **error/excepción no controlado** muestra una **respuesta genérica sin información técnica**

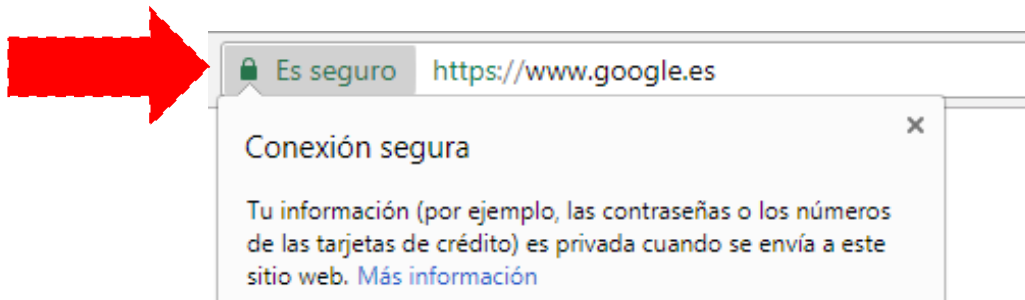
```
app.use( function (err, req, res, next ) {  
    console.log("Error producido: " + err);  
    if (! res.headersSent) {  
        res.send("Recurso no disponible");  
    }  
});
```

← Función de manejo de errores

```
app.listen(app.get('port'), function() {
```

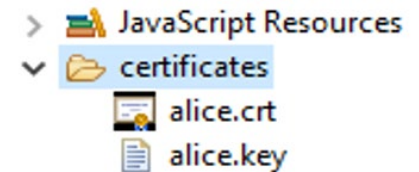
Https

- **Https** es un protocolo de transferencia **seguro** para hipertexto basado en http
- Cifra un canal de comunicación entre el servidor y navegador utilizando certificados SSL/TLS
 - Sí los datos son interceptados en ese canal, estos estarán cifrados
- Los navegadores dan información específica si una web usa https
 - Datos del certificado usado para cifrar (quien lo ha emitido)
 - Cualquiera puede emitir un certificado pero hay varias autoridades certificadoras confiables



Https

- Para agregar cifrado http incluimos los certificados en **una carpeta privada**
 - certificado.crt – certificado
 - certificado.key – clave
- Incluimos los módulos **https** y **fs** (filesystem) para procesamiento de ficheros



```
var fs = require('fs');  
var https = require('https');
```

- Modificamos la creación del canal **http** por -> **https**
 - Además del **listen** se debe incluir un **createServer()** que indica donde esta los certificados

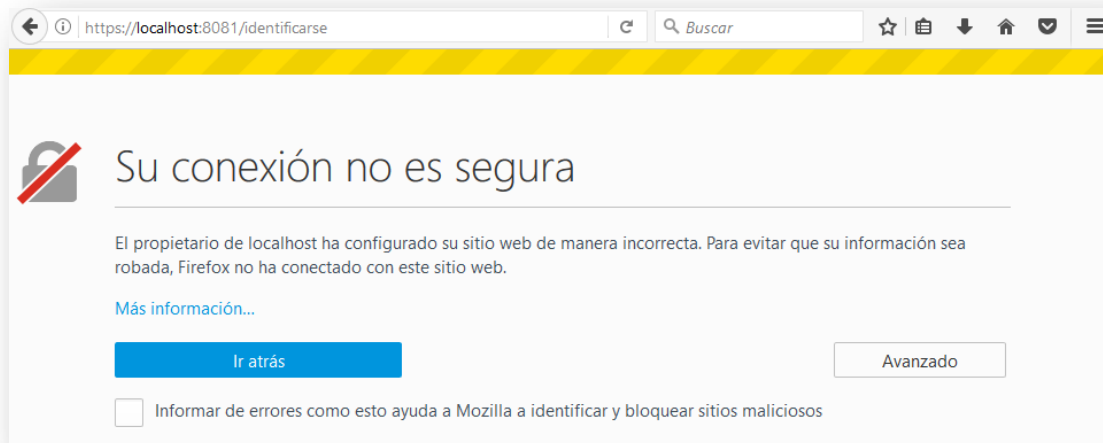
```
https.createServer({  
  key: fs.readFileSync('certificates/alice.key'),  
  cert: fs.readFileSync('certificates/alice.crt')  
}, app).listen(app.get('port'), function() {  
  console.log("Servidor activo");  
});
```

Https

- La aplicación ya usa https, las comunicaciones están cifradas
- Aunque el certificado no está emitido por una entidad confiable (lo hemos generado nosotros)
 - Nuestro navegador nos lo hará saber:



- Probablemente debemos agregar la página a excepciones de seguridad





Escuela de Ingeniería Informática

Escuela de Ingeniería Informática
School of Computer Science Engineering

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Sistemas Distribuidos e Internet

Tema 7 – Parte 2 Introducción a Node.js



Dr. Edward Rolando Núñez Valdez

nunezedward@uniovi.es