



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

### 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
- 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

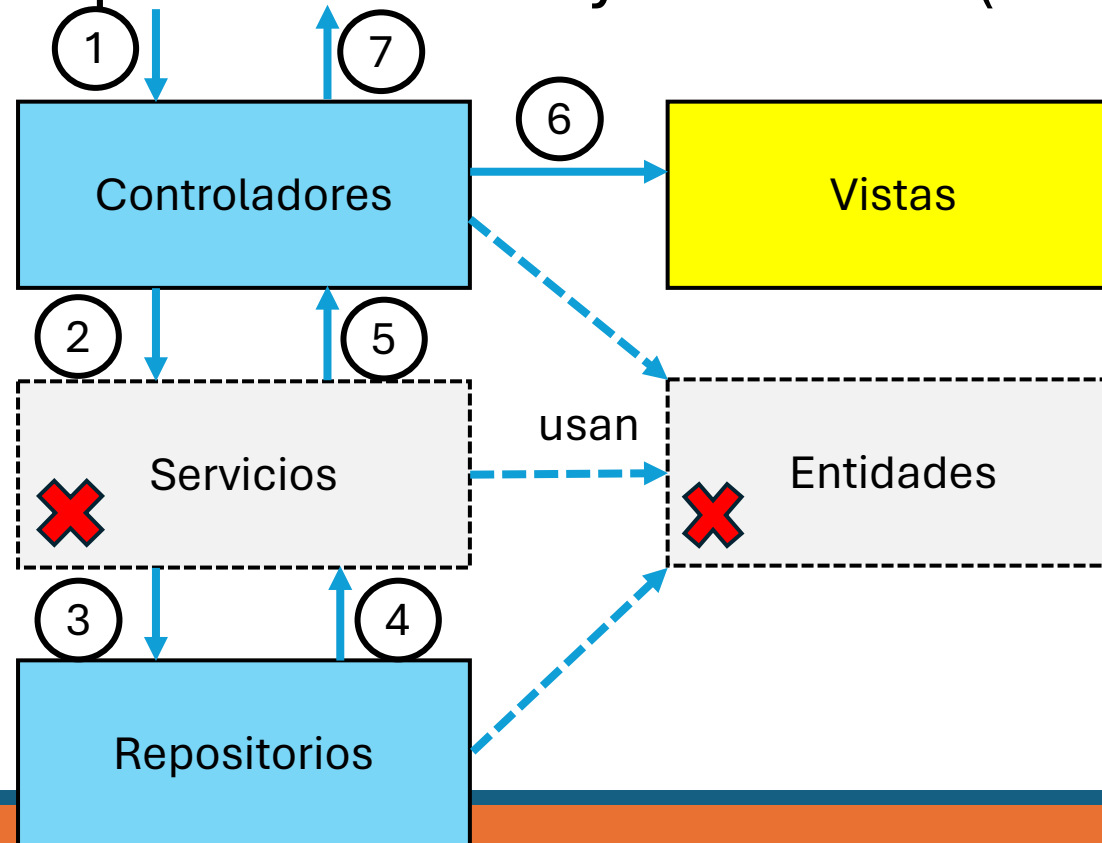
- Muchas aplicaciones no formalizan las entidades de forma estricta (con clases)
  - Una alternativa es **usar objetos**, donde resulta muy rápido modificar/añadir campos
    - Por ejemplo: objeto genérico canción (song) con 3 atributos

```
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)



# Arquitectura y módulos

- Los módulos pueden definirse como una **función, objeto o clase**
- Al incluir en la aplicación los módulos definidos como funciones, estas se ejecutan automáticamente.
  - ***require(<módulo>)***

- Módulo definido como una función

```
module.exports = function(app, twig) {  
  const limits = 100;  
  app.get("/publications", function(req, res) {  
    ...  
  });  
  app.get('/shopping', function (req, res) {  
    ...  
  })  
}
```

Declaración del módulo /routes/rsongs.js

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

Incluir el módulo /routes/rsongs.js

# Arquitectura y módulos

- Los módulos *definidos como objetos* permiten acceder a sus atributos y funciones con el punto (.)
  - 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

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

Declaración del módulo /modules/person.js

Incluir el módulo /modules/person.js



# Arquitectura y módulos

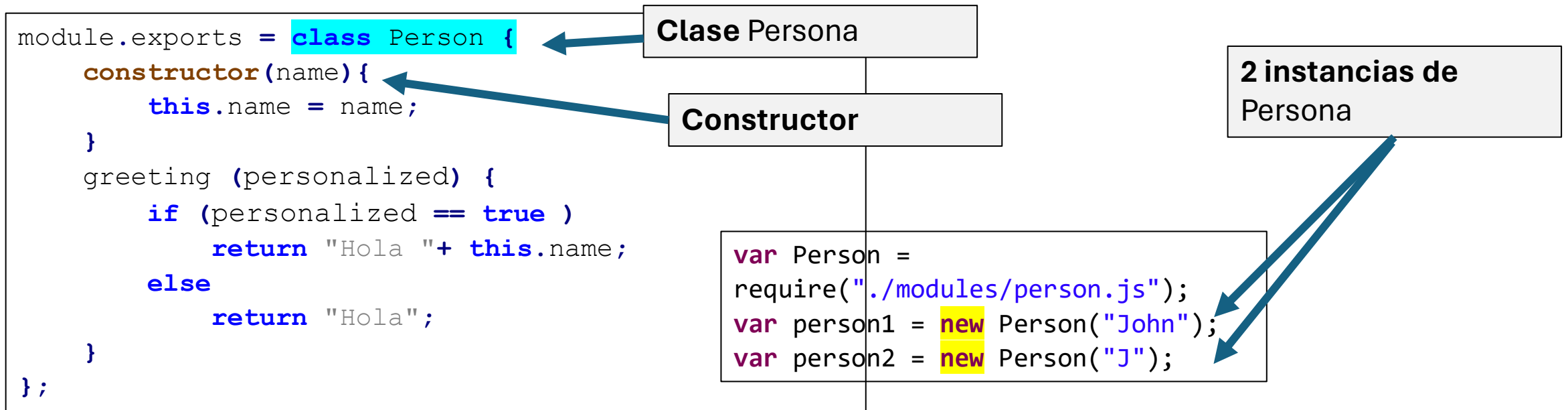
- La sintaxis de un objeto es muy distinta a la de una función

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

- **Un objeto no es lo mismo que una clase**
- 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 se define como **una clase** permiten crear instancias
  - Dentro de la clase, sus variables y funciones se referencian con ***this***



# Arquitectura y módulos

- Las aplicaciones utilizan módulos:
  - **Externos** descargados normalmente con el npm y contienen funcionalidad que puede ser común a muchas aplicaciones
    - body-parser, mongodb, swig, crypto, etc.
  - **Propios** implementación propia, normalmente específica o relativa a una aplicación
    - adsManager, adsRoutes, etc.
- Los módulos pueden ser usados:
  - Desde la aplicación Node.js-Express :
    - Ejemplo: el módulo body-parser es usado por la aplicación en el procesamiento del body (parámetros POST)
  - Desde otros módulos:
    - Ejemplo: el módulo adsRoutes utiliza el módulo adsManager para acceder a los anuncios y swig 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)
```

*users.js*

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

*payments.js*

- Caso 3: 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');
```

*users.js*

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

*payments.js*

# Arquitectura y módulos

- Usos de módulos integrados con la aplicación 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)
  - Ejemplos:
    - 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());
```

# Arquitectura y módulos

- Módulos con múltiples exportaciones
  - Si el módulo tiene varios submódulos, funciones o variables, se pueden importar de forma individual

```
const {check} = require('express-validator');
exports.songValidatorInsert = [
  check('title', 'Title is required').trim().not().isEmpty(),
]

exports.songValidatorUpdate = [
  check('title')
    .if(check('title').exists())
    .trim().isLength({min: 5})
    .withMessage('Title must be 5 or more characters')
]
```

*songsValidator.js*

```
const {songValidatorInsert} = require('./songsValidator')
```

*songsAPI.js*

# 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.)



# 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
- Código abierto

# Desventajas de MongoDB

- No es adecuada para aplicaciones con transacciones.
  - Manejo de transacciones ACID
    - Atomicity
    - Consistency -> Integridad
    - Isolation -> Aislamiento
    - Durability -> Persistencia
  - Soportado a partir de MongoDB 4.0.
- No soporta relaciones entre entidades para consulta.
- Tecnología joven.

# 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"
    }
  ]
}
```

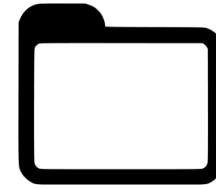
Valor: tipo simple

Valor: objeto { }

Valor: colección [ ]

# 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, momento de creación
    - 3 bytes: Identificador de la máquina
    - 2 bytes: PID – identificador del proceso
    - 3 bytes: Contador incremental

```
> db.proyectos.find()  
{ "_id" : ObjectId("574449da40fb278c24332fa6"), "nombre"  
  "opcion" : "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 cano
2016-10-16T18:00:04.716+0200 I FTDC [initandlisten] Initializing full-time diagnostic data
:/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

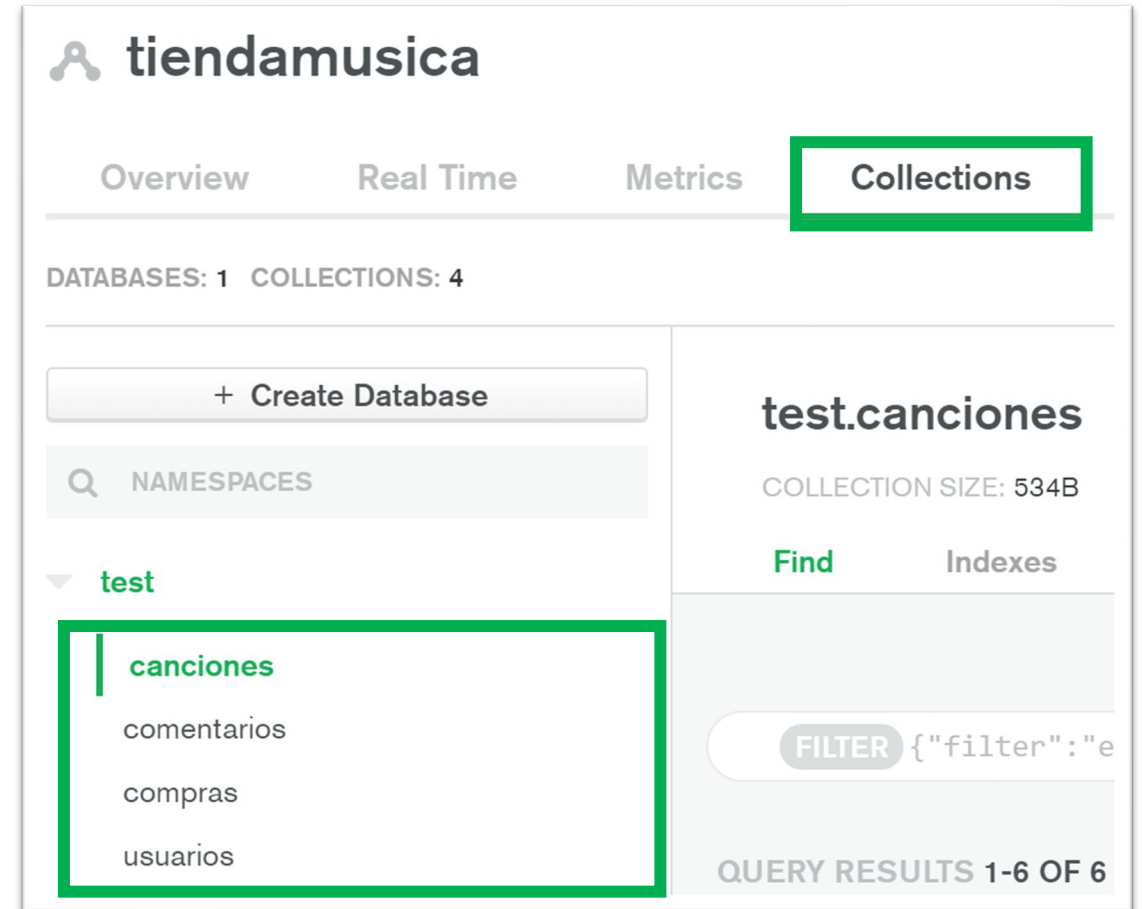
- Mongo en la Nube usando un proveedor de cloud computing
  - **MongoDB Atlas** (y otros muchos proveedores) permiten la creación de bases de datos en la Nube
  - Permite servidores “elásticos” pudiendo cambiar entre servidores con más o menos recursos según el uso requerido
  - Ofrece 512mb de almacenamiento 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://<dbuser>:<dbpassword>@ds229008.mlab.com:29008/movil>



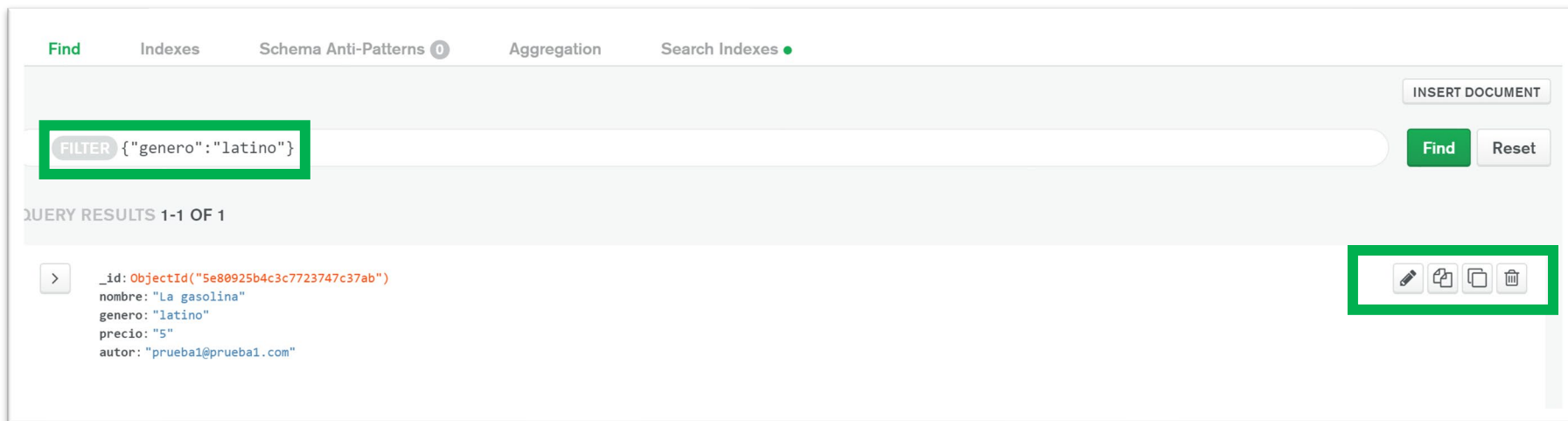
# Servidor de bases de datos MongoDB

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



# 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 a 6.X cada una utiliza API muy diferente

		RELEASE	DOCUMENTATION
		Next Driver	<a href="#">Reference</a>   <a href="#">API</a>
		6.4 Driver	<a href="#">Reference</a>   <a href="#">API</a>
		6.3 Driver	<a href="#">Reference</a>   <a href="#">API</a>
		6.2 Driver	<a href="#">Reference</a>   <a href="#">API</a>
4.4 Driver	<a href="#">Refer</a>	6.1 Driver	<a href="#">Reference</a>   <a href="#">API</a>
4.3 Driver	<a href="#">Refer</a>	6.0 Driver	<a href="#">Reference</a>   <a href="#">API</a>
4.2 Driver	<a href="#">Refer</a>	5.9 Driver	<a href="#">Reference</a>   <a href="#">API</a>
4.1 Driver	<a href="#">Refer</a>	5.8 Driver	<a href="#">Reference</a>   <a href="#">API</a>
4.0 Driver	<a href="#">Refer</a>	5.7 Driver	<a href="#">Reference</a>   <a href="#">API</a>
3.7 Driver	<a href="#">Reference</a>   <a href="#">API</a>		
3.6 Driver	<a href="#">Reference</a>   <a href="#">API</a>		

# Conexión a bases de datos MongoDB

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

```
npm install mongodb@4.1.1 --save
```



Especificar versión  
concreta


- 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();  
const {MongoClient} = require("mongodb")
```

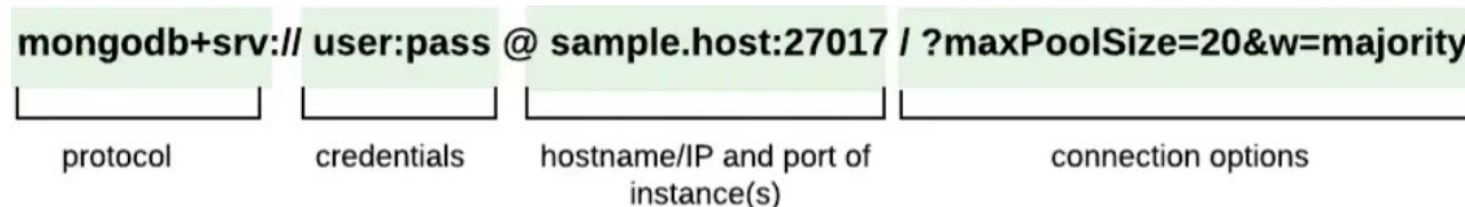
# Conexión a bases de datos MongoDB

- 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.

```
usersRepository.init(app, MongoClient);  
require("./routes/users.js")(app, usersRepository);
```



- Para conectarse a la base de datos debemos usar una **URL de conexión**:
  - Esta URL de conexión se puede guardar en una variable de entorno, fichero de configuración o en las **variables de la aplicación**
    - Esta última permitirá usarla en todos los módulos de la aplicación



```
app.set('connectionStrings', 'mongodb+srv://admin:<password>@tiendamusica.hy8gh.mongodb.net/  
myFirstDatabase?retryWrites=true&w=majority');
```

# Conexión a bases de datos MongoDB

- El módulo **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
      - **db** -> referencia a la base de datos, sobre este objeto se realizan las acciones (insertar, buscar, etc.)

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

*Ejemplo 1 con callback*

```
try {
  const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
  // realizar acciones (insertar, etc.)
  return result.insertedId;
} catch (error) {
  // Error al conectar
  throw (error);
}
```

*Ejemplo 2 con Async/Await*

# Gestión de datos con MongoDB

- En la función de callback (ejemplo1) usaremos el objeto **db** para gestionar los datos
  - **db.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

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

1º Fase

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

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

Sí respondemos aquí res.send(. . .)  
Más código

Cuidado!, la ejecución  
no es síncrona

Mongo

# Gestión de datos con MongoDB

- Segunda fase ejecución **asíncrona**

2º Fase connect

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

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

Más código
Más código
```

# Gestión de datos con MongoDB

- Tercera fase ejecución **asíncrona**

3º Fase insert

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

Respuesta final  
En caso de Éxito

# Gestión de datos con MongoDB

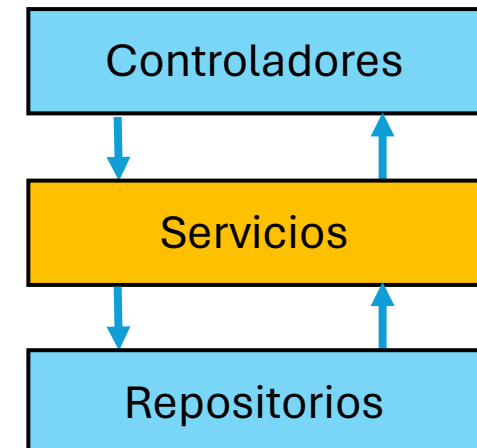
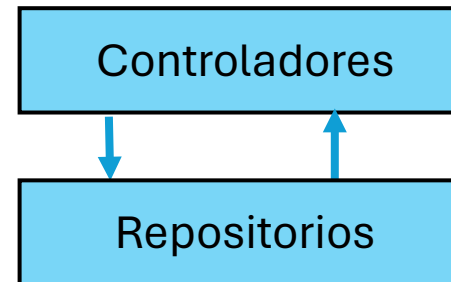
- Ejemplo **insertOne** usando promesas

```
...
insertSong: function (song, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), function
(err, db) {
        if (err) {
            funcionCallback(null);
        } else {
            let collection = db.collection('songs');
            //Con promesa
            collection.insertOne(song)
                .then(result => funcionCallback(result.ops[0]._id))
                .catch(err => funcionCallback(null));
            db.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: si 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('db'), function(err, db) {  
        ...  
        var collection = db.collection('songs');  
        collection.insert(song, function(err, result) {  
            if (err) {  
                funcionCallback(null);  
            } else {  
                funcionCallback(result.ops[0]._id);  
            }  
            db.close();  
        });  
    });  
});
```

Retorno Error

Retorno de Respuesta

# Arquitectura: acceso a datos

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

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

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

## Función de callback

Al acabar de **insertar**

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

```
funcionCallback(result.ops[0]._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). Timeouts, serialización, etc.
      - **writeConcern**. Un documento que expresa la inquietud de escritura.
        - `w:0`, `w:1`, `w:majority`, `w:tagSet`
        - Omitir el uso de la `writeConcern` de escritura predeterminada.
      - <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.
  - <https://docs.mongodb.com/v4.0/reference/method/db.collection.insertOne>

# Insertar un documento

- Ejemplo insertar usando promesas

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

Colección

Documento a insertar

insertedId -> \_id del documento insertado

# Insertar varios documentos

- 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` (opcional). Timeouts, serialización, etc.
      - **Ordered**: si es true, esta opción evita que se inserten documentos adicionales si uno falla.
      - **writeConcern**. Un documento que expresa la inquietud de escritura.
        - Omita el uso de la `writeConcern` de escritura predeterminada.
      - <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.
    - `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('db'), function (err, db) {
    if (err) {
      funcionCallback(null);
    } else {
      const collection = db.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():
    - consulta: 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 tan 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 (opcional). Timeouts, serialización, etc.
      - writeConcern. Un documento que expresa la inquietud de escritura.
        - Omitir el uso de la writeConcern de escritura predeterminada.
      - <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.
    - deletedCount: El número de de documentos borrado (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

```
const query = {name: "James" };
const collection = db.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 collection = db.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():
    - **Filtro**: selector para el documento a modificar.
    - **Documento**: nuevo documento que sustituye a los seleccionados por el criterio
    - **Upsert**: establecer la opción upsert a true para crear un nuevo documento si ningún documento coincide con el filtro.
    - \*Opciones (opcional). Timeouts, serialización, etc.
      - writeConcern. Un documento que expresa la inquietud de escritura.
      - <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.
    - **matchedCount**: número de documentos que coincidieron con el filtro.
    - **modifiedCount**: número de documentos que fueron modificados.
    - **upsertedCount**: número de documentos que fueron insertados.
    - **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 coincidad 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": "J",
  "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) unas 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 primitivo
  - El valor de este atributo es una **instancia de ObjectID**
  - En algunos casos, para acceder al valor como cadena: `<objeto>._id.toString()`
  - Ejemplo en JavaScript:

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


- Ejemplo en Swig o Twig:

```
<a href="/usr/{{ usr._id.toString() }}">
```

# 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 = new ObjectId(req.params.id);  
    var criterio = { "_id" : objectID };  
})
```

# Encriptación (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  
}  
dbManager 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

```
dbManager.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
```

- La función sesión se obtiene con **require**
  - La función puede recibir muchísimos parámetros de configuración opcionales. Algunos de los más comunes son:
    - **secret:** Es una clave secreta que se utiliza para firmar las cookies de sesión.
    - **resave:** (**true** / false) Indica si la sesión debe ser guardada de nuevo en el almacén de sesiones incluso **si no ha sido modificada** durante la solicitud.
    - **saveUninitialized:** (**true** / false) Indica si una sesión "no inicializada" debe ser guardada en el almacén de sesiones.


# 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

```
dbManager.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 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:

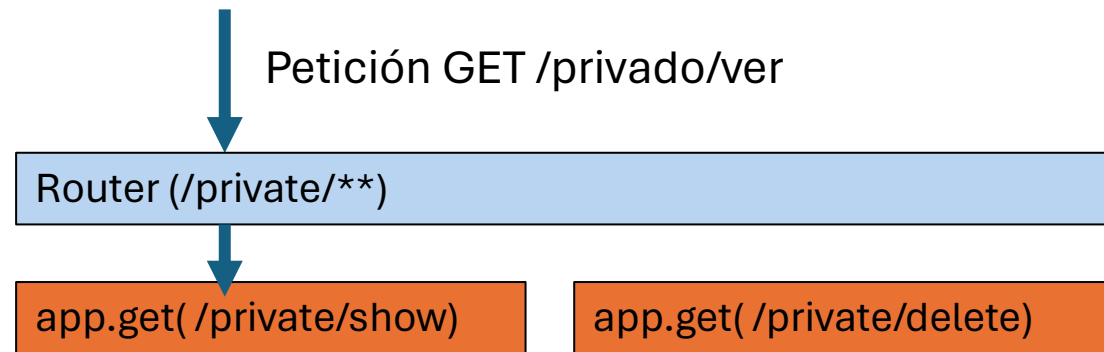
```
app.get('/private', function (req, res) {  
  if ( req.session.user == null) {  
    // NO autenticado!  
    res.redirect("/login");  
    return;  
  }  
  ...  
})
```

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 (middleware) 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

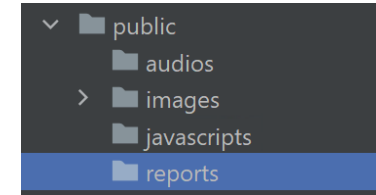
# Enrutadores

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

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

# Enrutadores

- Sí el orden no es correcto a la petición será respondida por quién 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)`

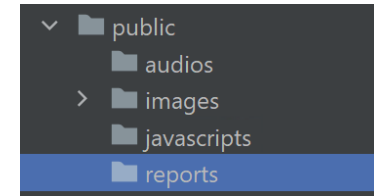
Responde a la petición  
se termina el ciclo

```
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)
```

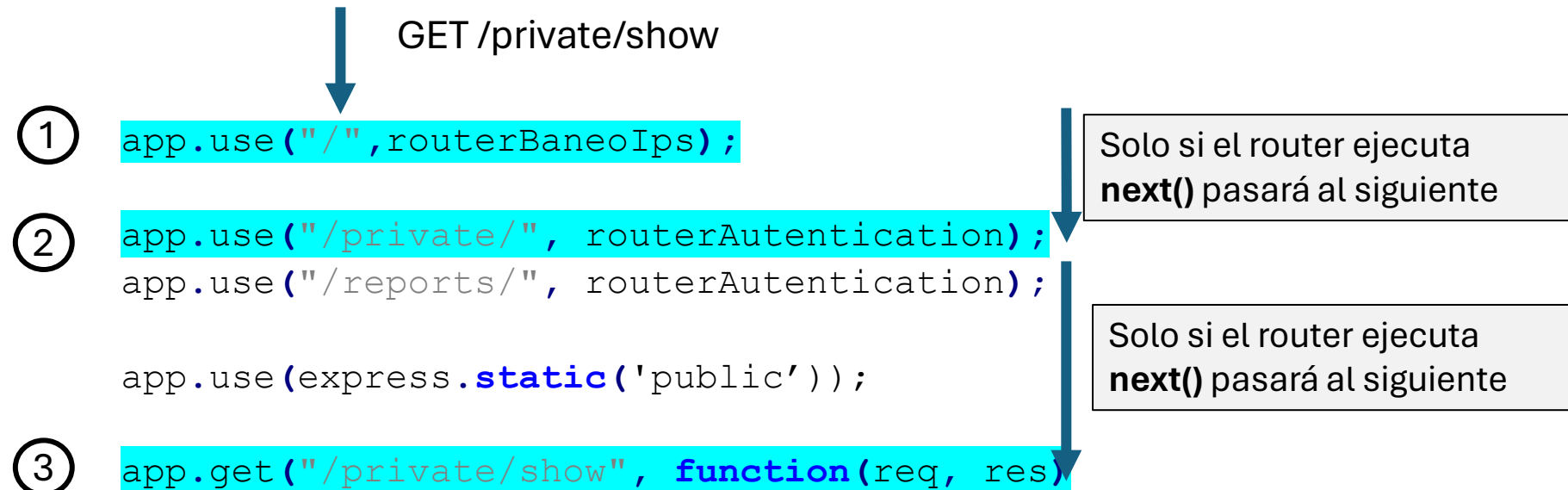
Responde a la petición  
se termina el ciclo

```
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





# 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 módulo externo ***express-fileupload***

```
npm install express-fileupload
```

- 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() )`

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

Asegurarse de que no es **null**

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

# Sistema de paginación



# Sistema de paginación

- Muchas aplicaciones utilizan sistemas de paginación
  - Tanto en las vistas como en la lógica de negocio
- La paginación puede **implementarse manualmente** o utilizando **elementos de un framework**
- Hay varios módulos que habilitan la paginación
  - express-paginate, express-pagination-middleware, otros.
  - El funcionamiento y configuración depende de cada módulo
- Vamos a ver como se **implementa un sistema de paginación**



# Sistema de paginación – Acceso a datos

- La paginación afecta a las consultas en base de datos
  - No se retornan todos los documentos de la colección
  - Se retornan los N correspondientes a una página
- Estableceremos un número de registros por página
  - Ejemplo: límite de 10 documentos por página
- Las funciones de consulta deben recibir:
  - Un parámetro con **el número página** para el que se solicitan los documentos

Número de página



```
obtenerSongsPg : function(criterio, pg, funcionCallback) {
```

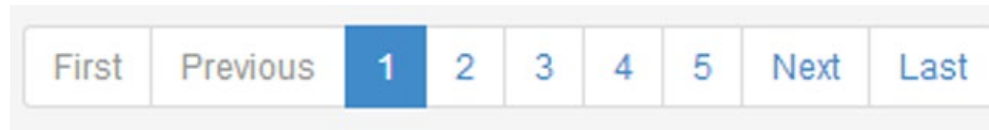
# Sistema de paginación – Acceso a datos

- Para afinar la consulta se aplican las funciones mongo:
  - **skip(int)** se mueve el cursor hacia delante, saltándose los n primeros documentos
  - **limit(int)** limita el número total de documentos
- El número de documentos a saltar (skip) depende de la página solicitada, por ejemplo:
  - Solicitan pg = 1 , skip = 0 no se salta ninguno
  - Solicitan pg = 2, skip = 10 , saltar los 10 de la página 1
  - Solicitan pg = 3, skip = 20 , saltar los 20 de la página 1 y 2
  - Formula para el skip = **skip( pg - 1 \* 10 )**
  - Máximo 10 documentos = **limit(10)**

```
collection.find(criterio).skip( (pg-1)*10 ).limit( 10 )
```

# Sistema de paginación – Acceso a datos

- También necesitamos el **número de documentos totales en la colección**
  - Determina el **número de páginas a mostrar**, por ejemplo: 80 documentos = 8 páginas



- Realizando un **count(función de callback con parámetros: err y cantidad)** se obtiene el número de documentos

```
var collection = db.collection('ads');  
collection.count(function(err, quantity) {  
  
}
```

Total de documentos en la colección anuncios

# Sistema de paginación – Controladores

- Las URLs relativas a colecciones paginables deben admitir:
  - Un parámetro opcional **pg**, en caso de omisión toma el valor 0
  - Por ejemplo:

```
app.get("/ads", function(req, res) {  
    // pg es String, ej "4" no 4  
    var pg = parseInt(req.query.pg);  
    // puede ser null  
    if ( req.query.pg == null){  
        pg = 1;  
    }  
})
```

- **Recordatorio:** los parámetros recibidos son **strings**, debemos pasarlo a **int**
- En esta implementación no contemplamos que pg no sea un número
  - En caso de error por ejemplo: **parseInt("Hola")** retorna **NaN**

# Sistema de paginación – Controladores

- El **controlador** debe preparar los **atributos del modelo** para:
  - Enviar los atributos a la vista
  - Que la vista muestra **la lista con el sistema de paginación**
- Se pueden seguir varios planteamientos:
  - **Planteamiento 1:** La vista determina que páginas hay que mostrar.
    - atributos del modelo:
      - Colección con los elementos de la página
      - Página actual
      - Número total de páginas
  - **Planteamiento 2:** El controlador determina que páginas hay que mostrar.
    - atributos del modelo:
      - Colección con los elementos de la página
      - Página actual
      - Colección con los números de las páginas a mostrar

# Sistema de paginación – Controladores

- Planteamiento 1:

```
var lastPage = total/10;  
// Sobran decimales  
if (total % 10 > 0 ){  
    lastPage = lastPage +1;  
}
```

Calcular la última página  
**Cuidado con la parte decimal**  
Sí 22 anuncios / 10 son : 2,2  
**La última página es la 3**  
**Sí hay decimales se añade una**

```
var respuesta = swig.renderFile('views/bshop.html',  
{  
    ads : ads,  
    currentPage : pg,  
    lastPage : lastPage  
});  
res.send(response);
```

Datos del modelo  
enviados a la vista

# Sistema de paginación – Vistas

- En la **vista** recomienda incluir al menos:
  - Notificación clara de la **página actual**
  - Acceso a las páginas **cercanas**
    - Por ejemplo: anterior y siguiente (si es que las hay)
  - Acceso a la **primera** y **última** página
- Un ejemplo de implementación
  - La plantilla utiliza swig para incluir enlaces a:
    - La **primera página** (pg=1) **siempre**
    - La **anterior a la actual** (pg=currentPage-1) **si es que existe** (> 0)
    - La **actual** (page=currentPage) **siempre**
    - La **siguiente a la actual** (page=actual+1) **si es que existe** (<= lastPage)
    - La **última** (pg= lastPage) **siempre**



# Sistema de paginación – Vistas

- Como el sistema lo hemos implementado nosotros le hemos dado valores lógicos a los atributos
  - **pg** empieza a contar en 1 (no en 0 como en Spring boot)
- Por ejemplo: vista del sistema de paginación

```
<!-- Primera -->
```

```
<a href="/ti?pg=1" >Primera</a>
```

```
<!-- Anterior (si la hay ) -->
```

```
{% if currentPage -1 >= 1 %}
```

```
    <a href="/ti?pg={{currentPage -1 }}" >{{currentPage -1 }}</a>
```

```
{% endif %}
```

```
<!-- Actual -->
```

```
<a href="/ti?pg={{currentPage }}" >{{currentPage }}</a>
```



# Sistema de paginación – Vistas

```
<!-- Siguiente (si la hay) -->
{% if currentPage +1 <= pgUltima %}
    <a href="/ti?pg={{currentPage+1 }}" > {{ccurrentPage+1 }}</a>
{% endif %}

<!-- Última -->
<a href="/ti?pg={{ lastPage }}" >Última</a>
```

- Se podrían no mostrar cuando dos enlaces se corresponde con la misma página, por ejemplo:
  - La **primera/ultima** coinciden con la **actual**
  - La **primera/ultima** coinciden con la **anterior** o la **siguiente**
  - **No esta claro si replicar enlaces perjudica la experiencia de usuario**

# 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

# 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 controlada** muestra una **respuesta genérica sin información técnica**

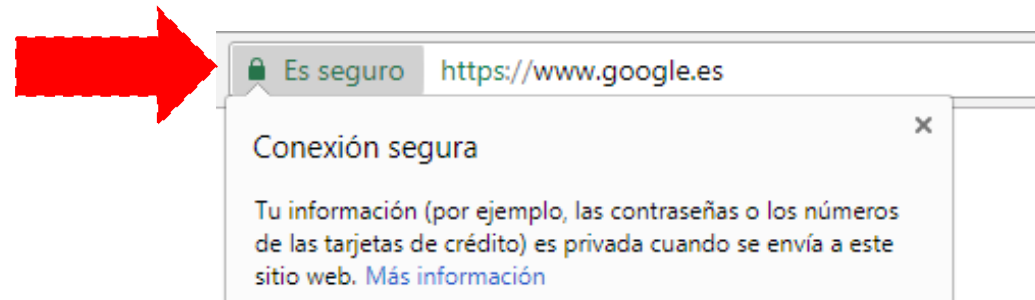
```
app.use( function (err, req, res, next ) {  
    console.log("Error producido: " + err);  
    if (! res.headersSent) {  
        res.send("Recurso no disponible");  
    }  
});
```

← Función de manejo de errores

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

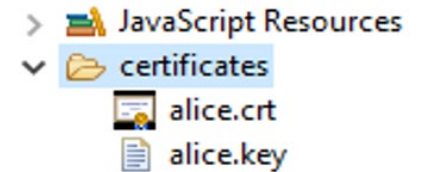
# Https

- **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 están los certificados

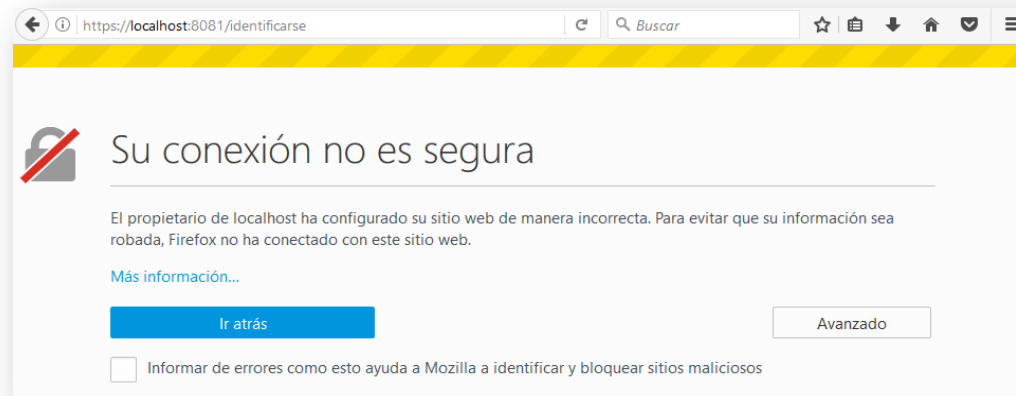
```
https.createServer({  
  key: fs.readFileSync('certificates/alice.key'),  
  cert: fs.readFileSync('certificates/alice.crt')  
}, app).listen(app.get('port'), function() {  
  console.log("Servidor activo");  
});
```

# 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 debamos 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 Introducción a Node.js



**Dr. Edward Rolando Núñez Valdez**

nunezedward@uniovi.es