

Sistemas Distribuidos e Internet

Tema 8

Node JS – Parte 2



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
- Sistema de paginación
- Captura de errores
- Https

Arquitectura y módulos

- La aplicación Node debería seguir una **arquitectura modular**
 - Dividendo las responsabilidades en diferentes módulos
 - Estos módulos pueden comunicarse entre sí
- Comparandolo 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 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 basicamente llamadas a repositorios)

Arquitectura y módulos

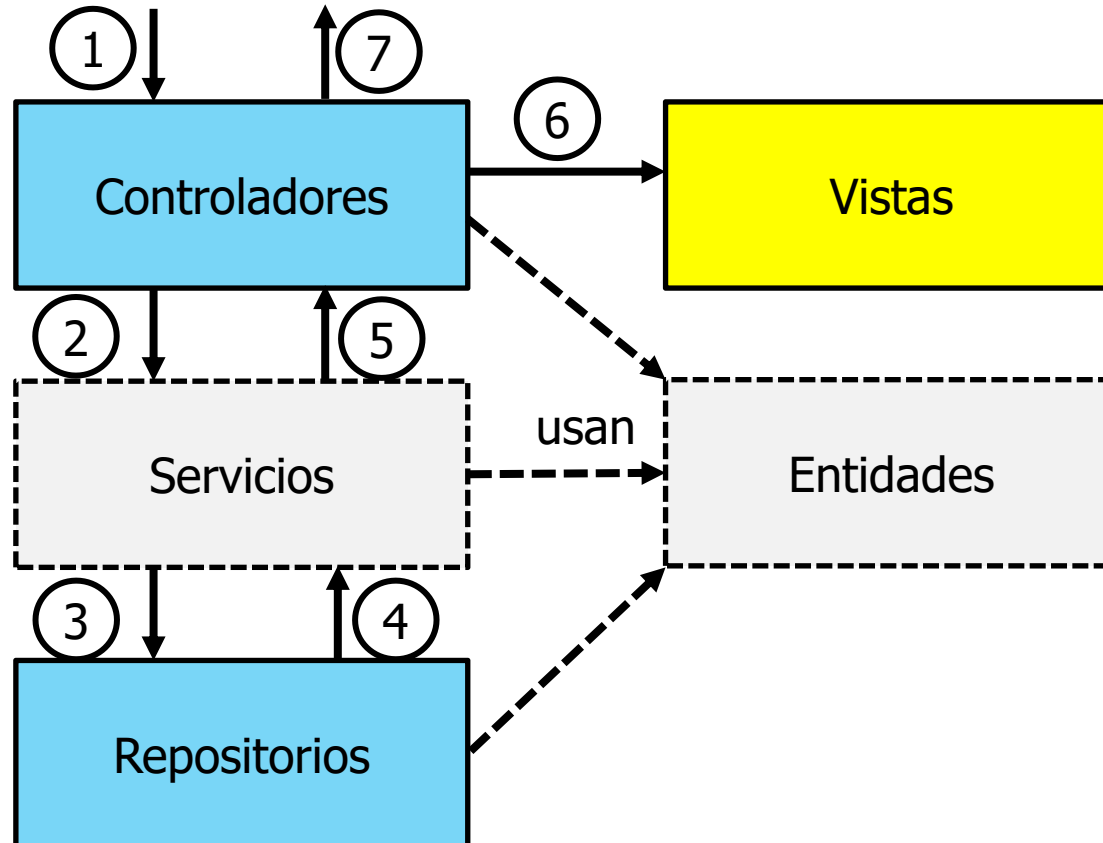
- Muchas aplicaciones no formalizan las entidades de forma estricta (con clases)
 - Una alternativa es usar **objetos**, donde resulta muy rápido modificar/añadir campos
 - Ej: objeto genérico canción con 3 atributos

```
var cancion = {  
  nombre : req.body.nombre,  
  genero : req.body.genero,  
  precio : req.body.precio  
}
```

- 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 incluir 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>)`)**
- Ej, módulos que agregan rutas a la aplicación



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

Declaración del módulo /routes/rcanciones.js

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

Incluir el módulo /routes/rcanciones.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 = {  
  nombre : null,  
  apellidos : null,  
  init : function(nombre) {  
    this.nombre = nombre;  
  },  
  saludar : function(personalizado) {  
    if ( personalizado == true )  
      return "Hola " + this.nombre;  
    else  
      return "Hola";  
  }  
};
```

init nombre común
para el inicializador

this.nombre es la variable
nombre del objeto

Declaración del módulo /modules/persona.js

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

Incluir el módulo /modules/persona.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 Persona {  
  constructor(nombre) {  
    this.nombre = nombre;  
  }  
  saludar (personalizado) {  
    if ( personalizado == true )  
      return "Hola " + this.nombre;  
    else  
      return "Hola";  
  }  
};
```

Clase Persona

Constructor

```
var Persona = require("./modules/persona.js");  
var persona1 = new Persona("John");  
var persona2 = new Persona("J");
```

2 instancias de
Persona

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
 - gestorAnuncios, rutasAnuncios, etc.
- Los módulos deben ser usados desde:
 - **La aplicación express** : Ej, el módulo **body-parser** es usado por la aplicación en el procesamiento de los body (parámetros POST)
 - **Otros módulos**: Ej, el módulo **rutasAnuncios** utiliza el módulo **gestorAnuncios** para acceder a los anuncios y **swig** para generar respuestas basadas en plantillas.

Arquitectura y módulos

- Módulos estándar

- Para poder utilizarlos desde otros módulos o partes de la app se puede optar por varias alternativas, algunas de ellas son:

- Obtener el objeto/función allí donde sea requerido (alternativa no muy mantenible, los cambios pueden ser costosos)

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

usuarios.js

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

pagos.js

- Obtener el objeto/función una vez y enviarlo como parámetro a otros módulos

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

app.js

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

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

- 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("utilidades.js"));
```

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

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

Arquitectura y módulos

- Módulos integrados con la aplicación Express
 - Suelen ser módulos **vinculados a express**
 - Se obtiene el objeto/función correspondiente al módulo
 - En algunos casos se puede configurar
 - 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)
 - Ej:
 - 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

- Son bases de datos **no relacionales**
 - 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**:

```
{
  nombre: "Cambiar ordenadores",
  equipos: 3,
  atendido: true,
  descripcion: "Cambiar todos los ordenadores",
  detalles: {
    categoria: "mantenimiento",
    coste: 4233
  },
  incidencias: [
    {
      descripcion: "Inicio sin problemas",
      fecha: "23-06-2016"
    },
    {
      descripcion: "Falta de material",
      fecha: "24-06-2016"
    }
  ]
}
```

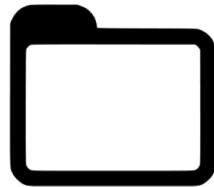
Valor: tipo simple

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

```
{  
  "nombre" : "John",  
  "apellido" : "Doe",  
  "calidad" : 10  
}
```

```
{  
  "centro" : "uniovi",  
  "calidad" : 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.insert**({ documento }): insertar un nuevo documento
 - **Colección.update**({criterio de selección } , { nuevo documento }): actualizar documentos
 - **Colección.remove**({criterio de selección }): eliminar documentos
 - Otros.

Bases de datos MongoDB

- Al salvar un documento, Mongo 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 Máquina : identificador
 - 2 PID : proceso
 - 3 Parte 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 daos, Ej **C:\data\db**
 - Acceder a la carpeta donde se instalo **\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 **mLab**
 - **mLab** (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 **500mb** de almacenamiento de datos sin coste
 - Permite crear múltiples bases de datos

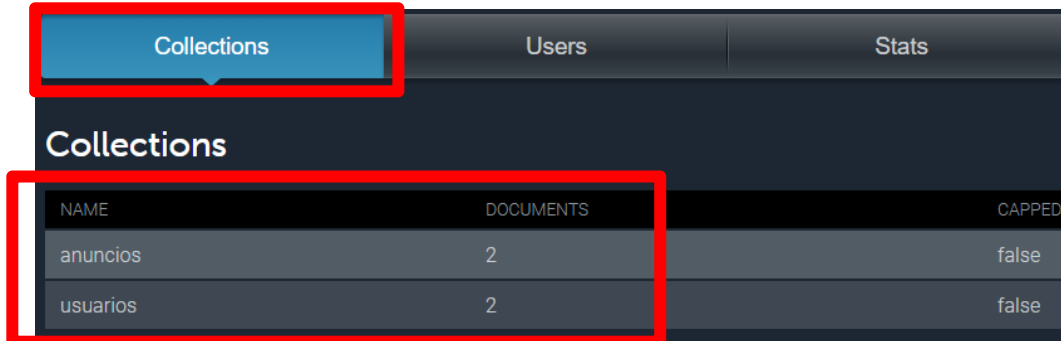
>	NAME	RAM	SIZE ?	SIZE ON DISK ?
>	✓ ds229008/movil	shared	16.98 KB	16.00 MB
>	✓ ds117935/tiendamusica	shared	24.84 KB	16.00 MB

- Por **seguridad** requiere la creación de un **usuario-password** para la base de datos
- Obtenemos una cadena de conexión, EJ:

<mongodb://<dbuser>:<dbpassword>@ds229008.mlab.com:29008/movil>

Servidor de bases de datos MongoDB

- Desde **mLab** se puede consultar todas las **colecciones** y **documentos** que se van creando.
 - EJ, colecciones: **anuncios** y **usuarios**, ambas con dos documentos



Collections		
NAME	DOCUMENTS	CAPPED?
anuncios	2	false
usuarios	2	false

- Se puede ver el **contenido de los documentos**, realizar **búsquedas**, **borrarlos** y **modificarlos**



```
{
  "_id": {
    "$oid": "5a7e105582fdc40c0589804d"
  },
  "descripcion": "PRueba",
  "precio": 2323
}
```

Conexión a bases de datos MongoDB

- Para que una aplicación Node 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 Nodo
 - <https://mongodb.github.io/node-mongodb-native/>
 - No está incluido en el core de Node
- Actualmente se mantienen dos versiones release de **mongodb**: 2.X y 3.X cada una utiliza API muy diferente

Releases	
RELEASE	DOCUMENTATION
3.0 Driver	Reference API
2.2 Driver	Reference API

Conexión a bases de datos MongoDB

- Utilizaremos la release 2.2 . En la instalación del módulo se debe especificar la versión.

Especificar versión concreta

`npm install mongodb@2.2.33 --save`

- Si no especificamos versión instala la que “considera la ultima”
- Agregamos el módulo **mongo** a la aplicación, con **require**

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


Conexión a bases de datos MongoDB

- El módulo **mongo** 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/rcanciones.js")(app, swig, mongo);
```



- Para conectarse a la base de datos podemos usar una **URL de conexión**, Ej:
 - `mongodb://<dbuser>:<dbpassword>@ds229008.mlab.com:29008/movil`
- Es recomendable guardar la URL en las **variables de la aplicación**
 - Así se podrá usar en todos los módulos de la aplicación

```
app.set('port', 8081);
```

```
app.set('db', 'mongodb://<dbuser>:<dbpassword>@ds229008.mlab.com:29008/movil');
```

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
 - **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.)  
    }  
});
```

Gestión de datos con MongoDB

- Usaremos el objeto **db** para gestionar los datos
 - **db.collection(<nombre colección>)** da acceso a la 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, Ej:
 - **insert(objeto JSON, función manejadora(err, resultado))** -> para guardar un nuevo documento
 - El **resultado** de la función manejadora depende de la **acción**, Ej:
 - 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

Gestión de datos con MongoDB

■ Ejemplo **insert**


```
var cancion = {
  nombre : req.body.nombre,
  genero : req.body.genero
}
mongo.MongoClient.connect(app.get('db'), function(err, db) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = db.collection('canciones');
    collection.insert(cancion, 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 las funciones **manejadoras**



1º Fase

```
var cancion = {
  nombre : req.body.nombre,
  genero : req.body.genero
}

mongo.MongoClient.connect(app.get('db'), function(err, db) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = db.collection('canciones');
    collection.insert(cancion, function(err, result) {
      if (err) {
        res.send("Error al insertar " + err);
      } else {
        res.send("Agregada ");
      }
    });
    db.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 cancion = {
  nombre : req.body.nombre,
  genero : req.body.genero
}

mongo.MongoClient.connect(app.get('db'), function(err, db) {
  if (err) {
    res.send("Error de conexión: " + err);
  } else {
    var collection = db.collection('canciones');
    collection.insert(cancion, 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

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

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

Más código
Más código

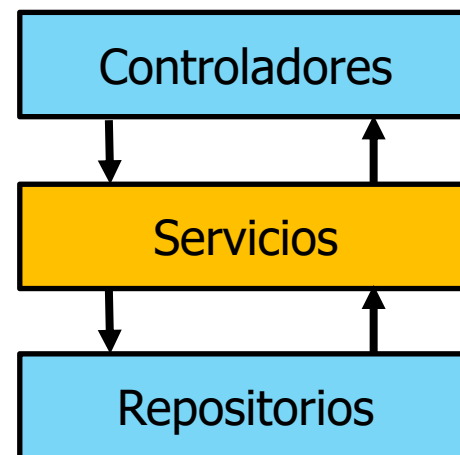
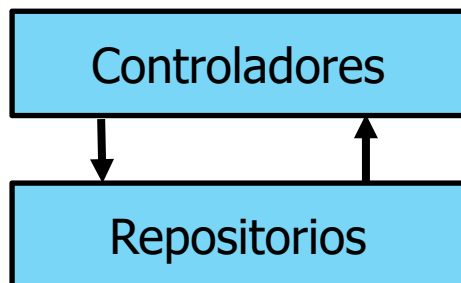
3º Fase insert



Respuesta final
En caso de Éxito

Arquitectura: acceso a datos

- Debemos encapsular el acceso a datos en **uno o varios módulos**
 - Dependiendo del número de **entidades** y **operaciones**, 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**
 - Ej: 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, Ej para enviarle el **id** del objeto insertado.

```
insertarCancion : function(cancion, funcionCallback) {  
  this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {  
    ...  
    var collection = db.collection('canciones');  
    collection.insert(cancion, function(err, result) {  
      if (err) {  
        funcionCallback(null);  
      } else {  
        funcionCallback(result.ops[0]._id);  
      }  
      db.close();  
    });  
  });  
});
```

The diagram illustrates the callback function's role in handling asynchronous database operations. Two callout boxes with arrows point to specific lines in the code:

- A box labeled "Retorno Error" points to the line `funcionCallback(null);` inside the `if (err)` block.
- A box labeled "Retorno de Respuesta" points to the line `funcionCallback(result.ops[0]._id);` inside the `else` block.

Arquitectura: acceso a datos

- Ej: uso de un módulo de acceso a datos desde un controlador
 - **gestorDB** es la referencia al módulo

```
app.post("/cancion", function(req, res) {
    var cancion = {
        nombre : req.body.nombre,
        genero : req.body.genero
    }
    gestorBD.insertarCancion(cancion, 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);
```



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

Insert

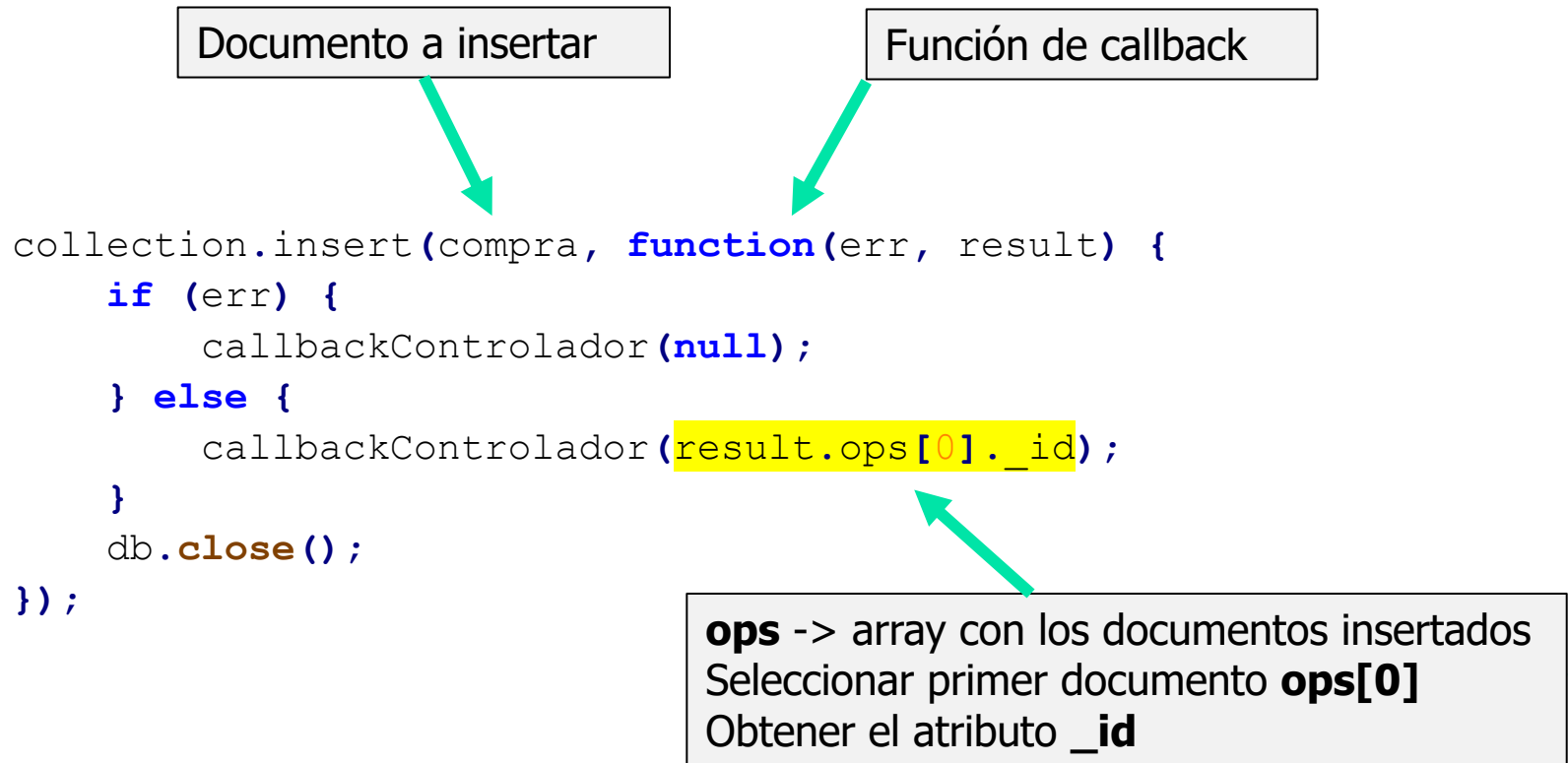
- Inserta un documento o array de documentos en una colección
- Si los documentos no contienen el campo **_id** lo agrega automáticamente
- Parámetros de **insert**:
 - Documento o documentos a insertar
 - *Opciones (opcional). Timeouts, serialización, etc.
 - <http://mongodb.github.io/node-mongodb-native/2.2/api/Collection.html#insert>
 - Función de callback, con parámetros, **err** y **insertWriteOpsResult**:
 - **insertWriteOpsResult** contiene varios atributos
 - **insertedCount** -> número de documentos insertados
 - **ops** -> array con todos los documentos insertados
 - **insertedIds** -> array con solo las **_id** de los documentos insertados
 - **connection** -> referencia al objeto conexión utilizado para realizar la operación
 - **result** -> respuesta generada por la base de datos (puede cambiar dependiendo de la versión {**"ok"** : 1 , **"n"** : 1}
ok :1 = operación con éxito, n : 1 = 1 documento insertado

Resultados
más utilizados

La mayor
parte no se
suelen utilizar

Insert

- Ejemplo insertar



Remove

- Elimina uno o varios documentos de una colección
- Parámetros de **remove**:
 - Criterio : selector para la operación de eliminar
 - {"tipo" : "casa"} = los documentos de tipo casa
 - {"precio":{ \$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 : [{"edad" : 20},{ "edad" : 30},{ "edad":40}]} = documentos con edad 20, 30 o 40
. **OR**
 - {\$and : [{"edad" : 40},{ "empresa" : "CSC"}]} = documentos con edad 40 y empresa CSC
. **AND**
 - *Opciones (opcional). Timeouts, serialización, etc.
 - Función de callback, con parámetros, **err** y **WriteOsResult** .
 - **WriteOsResult** :
 - **result** -> respuesta generada por la base de datos (puede cambiar dependiendo de la versión) {"ok" : 1 , "n" : 1}
ok :1 = operación con éxito, n : 1 = 1 documento eliminado

Más utilizado
result.n

Remove

- Ejemplo remove

```
collection.remove({ nombre : "J"}, function(err, obj) {  
    if (err) {  
        callbackControlador(null);  
    } else {  
        callbackControlador(obj.result.n);  
    }  
    db.close();  
});
```



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

Update

- Actualiza uno o varios documentos de una colección
- Parámetros de **update**:
 - Criterio : selector para documento o documentos a modificar
 - Documento: nuevo documento que sustituye a los seleccionados por el criterio
 - *Opciones (opcional). Timeouts, serialización, etc.
 - Función de callback, con parámetros, **err** y **WriteOsResult** .
 - **WriteOsResult** :
 - **result** -> respuesta generada por la base de datos (puede cambiar dependiendo de la versión) {"ok" : 1 , "n" : 1}
 - ok :1 = operación con éxito, n : 1 = 1 documento eliminado

Más utilizado
result.n

Update

- Ejemplo update
 - Sustituye **completamente** los documentos que cumplen el criterio de selección por el nuevo documento

```
var criterio = { nombre : "J"};
var nuevaPersona = { nombre : "R", apellido : "R"};
collection.update(criterio,nuevaPersona, function(err, obj) {
  if (err) {
    callbackControlador(null);
  } else {
    callbackControlador(obj);
  }
  db.close();
});
```

Pre-update	Post-update
<pre>{ "nombre" : "J", "apellido" : "J", "país" : "es", "idioma" : "es" }</pre>	<pre>{ "nombre" : "R", "apellido" : "R" }</pre>

Update

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

```
var criterio = { nombre : "J"};
var atributos = { apellido: "R", edad : 40};
collection.update(criterio, {$set: atributos}, function(err, obj) {
  if (err) {
    callbackControlador(null);
  } else {
    callbackControlador(obj);
  }
  db.close();
});
```

Pre-update

```
{
  "nombre" : "J",
  "apellido" : "J",
  "país" : "es",
  "idioma" : "es"
}
```

Post-update

```
{
  "nombre" : "J",
  "apellido" : "R",
  "edad" : 40,
  "país" : "es",
  "idioma" : "es"
}
```

Find

- Realiza una búsqueda de documentos por criterio
- find() no tiene un parámetro con función de callback
- El criterio selector, se expresa en formato JSON / objeto , EJ:
 - {} = todos los documentos
 - {"tipo" : "casa"} = los documentos de tipo casa
 - {"tipo" : "casa", "metros" : 100} = los documentos de tipo casa y metros 100
 - Es equivalente a utilizar un AND
 - {"precio":{ \$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 : [{"edad" : 20},{ "edad" : 30},{ "edad":40}]} = documentos con edad 20, 30 o 40 . **OR**
 - {\$and : [{"edad" : 40},{ "empresa" : "CSC"}]} = documentos con edad 40 y empresa CSC . **AND**

Find

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

Resultado: array documentos




```
collection.find({}).toArray(function(err, usuarios) {  
    if (err) {  
        callbackControlador(null);  
    } else {  
        callbackControlador(usuarios);  
    }  
    db.close();  
});
```

Find

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

Filtra el resultado antes de obtener el array



```
collection.find(criterio).limit(3).toArray(function(err, usuarios) {  
    if (err) {  
        funcionCallback(null);  
    } else {  
        funcionCallback(usuarios);  
    }  
    db.close();  
});
```

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 tiene un **_id : ObjectId**

- **ObjectID** no es un tipo simple
- El valor de este atributo es una **instancia de ObjectId**
- Para acceder al valor como cadena: <objeto>._id.toString()

- Ej en JS:

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


- Ej en swig:

```
<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 = gestorBD.mongo.ObjectId(req.params.id);  
    var criterio = { "_id" : objectID };
```

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 otra información 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";  
encriptador = crypto.createHmac('sha256', secreto);
```

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

```
valorCifrado = encriptador.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
 - A bajo nivel usan tecnología
 - Ej: 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("/identificarse", 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  
}  
gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
  
});
```

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

```
gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
    if (usuarios == null || usuarios.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 **mail = 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()**
- Ej, 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, ej : **email**

```
gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
    if (usuarios == null || usuarios.length == 0) {  
        req.session.usuario = null;  
        // respuesta no autenticado  
    } else {  
        req.session.usuario = usuarios[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('/desconectarse', function (req, res) {  
  req.session.usuario = 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**. EJ:

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

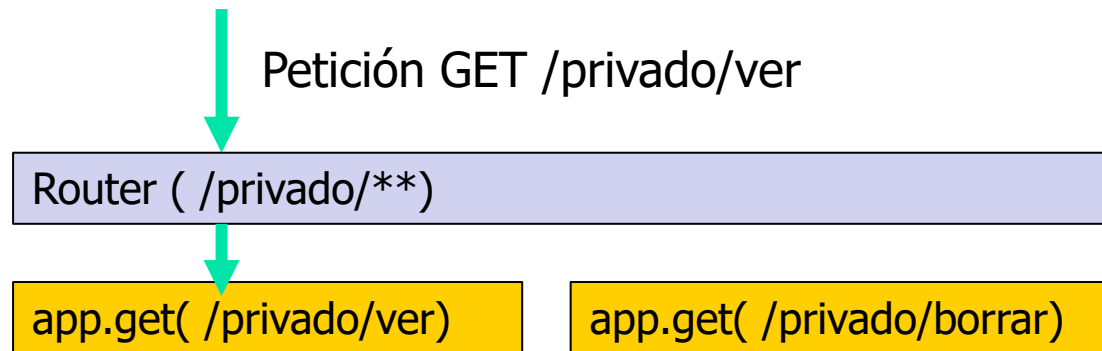


Si **usuario == null** no hay
usuario autenticado
No puede entrar en **/privado**

- 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 (Ej, redireccionándola)

Enrutadores

- El 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
- Ej, creación de un enrutador

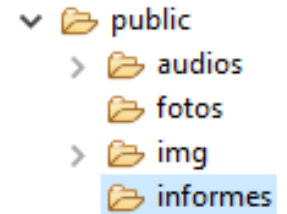
```
var routerAutenticacion = express.Router();
routerAutenticacion.use(function(req, res, next) {
  if ( req.session.usuario )
    // Hay usuario autenticado
    next();
  else
    // No hay usuario autenticado
    res.redirect("/login");
});
```

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
- Ej:



```
app.use("/privado/",routerAutenticacion);  
app.use("/informes/",routerAutenticacion);
```

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

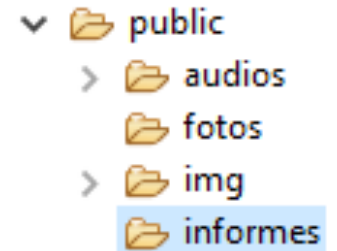
```
app.get("/privado/ver", function(req, res) app.get  
app.get("/privado/borrar", function(req, res) app.get  
...
```

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 critico
- El orden determina quien responderá a la petición:
- La función **next()** de los routers deja continuar la petición

GET /privado/ver



① `app.use("/privado/",routerAutenticacion);`
`app.use("/informes/",routerAutenticacion);`

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

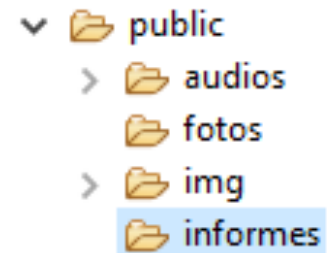
② `app.get("/privado/ver", function(req, res)`
`app.get("/privado/borrar", function(req, res)`
`...`

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

Enrutadores

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

GET /privado/ver



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

① `app.get("/privado/ver", function(req, res)`
`app.get("/privado/borrar", function(req, res)`

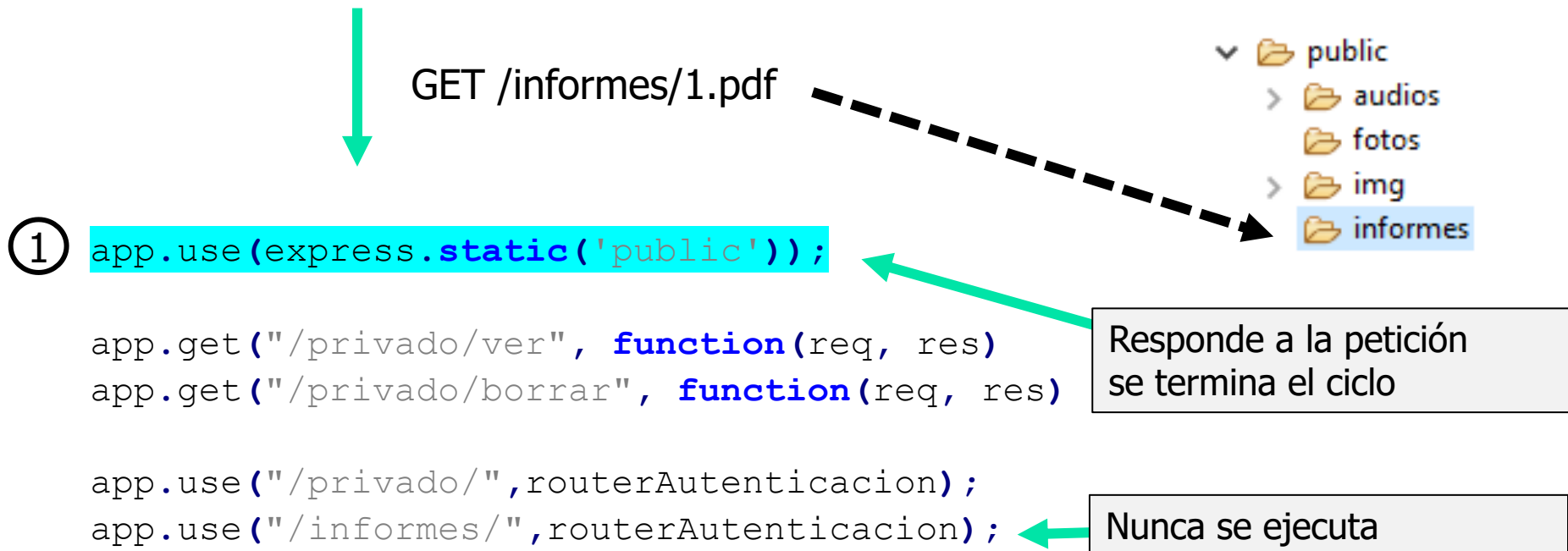
Responde a la petición
se termina el ciclo

```
app.use("/privado/",routerAutenticacion);  
app.use("/informes/",routerAutenticacion);
```

Nunca se ejecuta

Enrutadores

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



Enrutadores

- Se pueden **aplicar multiples enrutadores** sobre un mismo path
- Se delega una única acción en cada enrutador, Ej:
 - routerAutenticación -> comprueba si hay usuario en sesión
 - routerBaneoIPs -> comprueba si la IP está en una lista negra

GET /privado/ver

① `app.use("/", routerBaneoIps);`

② `app.use("/privado/", routerAutenticacion);`
`app.use("/informes/", routerAutenticacion);`

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

③ `app.get("/privado/ver", 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
- Ej, 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:
 - `/usuario/anuncio/ver/:id`
 - `/usuario/anuncio/reservar/:id`
 - `/usuario/propietario/anuncio/modificar/:id`
 - `/usuario/propietario/anuncio/eliminar/:id`
- Donde habría dos enrutadores
 - `routerUsuario` -> comprueba que hay un usuario en sesión
 - `routerPropietario` -> comprueba que es el propietario del anuncio

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>**
 - Ej: El formulario incluye el input de tipo fichero con clave foto
 - Debemos recordar el **enctype** de tipo **multipart/form-data**

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

- Procesamiento del 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 **file.mv(<directorio>,callback())**

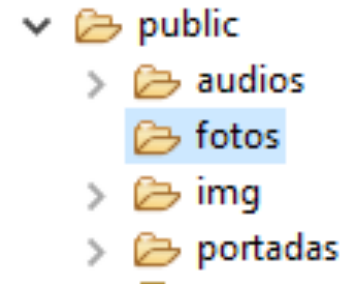
```
if (req.files.foto != null) {  
    var ficheroFoto = req.files.foto;  
    ficheroFoto.mv('public/fotos/' + id + '.jpg', function(err) {  
        if (err) {  
            // ERROR  
        } else {  
            // EXITO  
        }  
    });  
}
```

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 (ej ID del usuario)
- En algunos casos es necesario hacer **comprobaciones de autorización** para acceder a ficheros de **directorio de acceso web**
 - **/static/informes** solo pueden acceder usuarios registrados.
 - **/static/fotos/31** solo puede acceder el usuario 31




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
 - Ej: 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



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

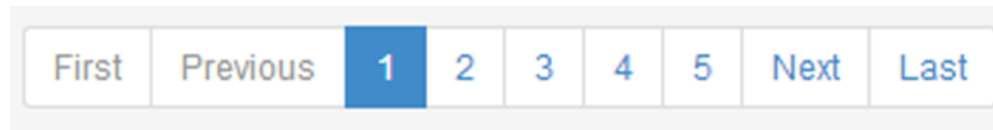
Sistema de paginación – Acceso a datos

- 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 escapar depende de la página solicitada, Ej:
 - 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**, Ej 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('anuncios');  
collection.count(function(err, cantidad) {  
  
}
```



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
 - Ej:

```
app.get("/anuncios", 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 ej, **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, Ej:
 - **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

■ Ej, planteamiento 1

```
var pgUltima = total/10;  
// Sobran decimales  
if (total % 10 > 0 ){  
    pgUltima = pgUltima+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/btienda.html',  
{  
    anuncios : anuncios,  
    pgActual : pg,  
    pgUltima : pgUltima  
});  
res.send(respuesta);
```

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**
 - Ej: 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=actual-1) **si es que existe** (> 0)
 - La **actual** (page=actual) **siempre**
 - La **siguiente a la actual** (page=actual+1) **si es que existe** ($\leq \text{ultimaPagina}$)
 - La **última** (pg=ultimaPagina) **siempre**

Primera	1	2	3	Última
---------	---	---	---	--------

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)
- Ej, vista del sistema de paginación

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

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

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

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

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

```
{% endif %}
```

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

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

Sistema de paginación – Vistas

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

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

- Se podrían no mostrar cuando dos enlaces se corresponde con la misma página, Ej:
 - 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**

Captura de errores

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

```
Error: Argument passed in must be a single String  
    at new ObjectID (C:\Users\jordansoy\work\Tiend  
    at Function.ObjectID (C:\Users\jordansoy\work\
```

- 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 OWSAP (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(funcion)** como elemento final de la aplicación
 - Sí detecta un **error/excepción no controlado** muestra una **respuesta genérica sin información técnica**

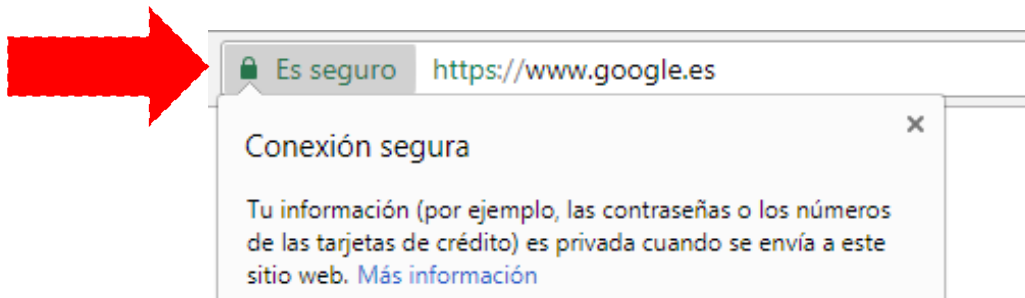
```
app.use( function (err, req, res, next ) {  
    console.log("Error producido: " + err);  
    if (! res.headersSent) {  
        res.send("Recurso no disponible");  
    }  
});
```

← Función de manejo de errores

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

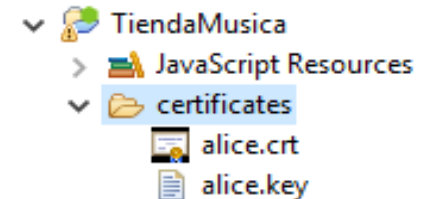

Https

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



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

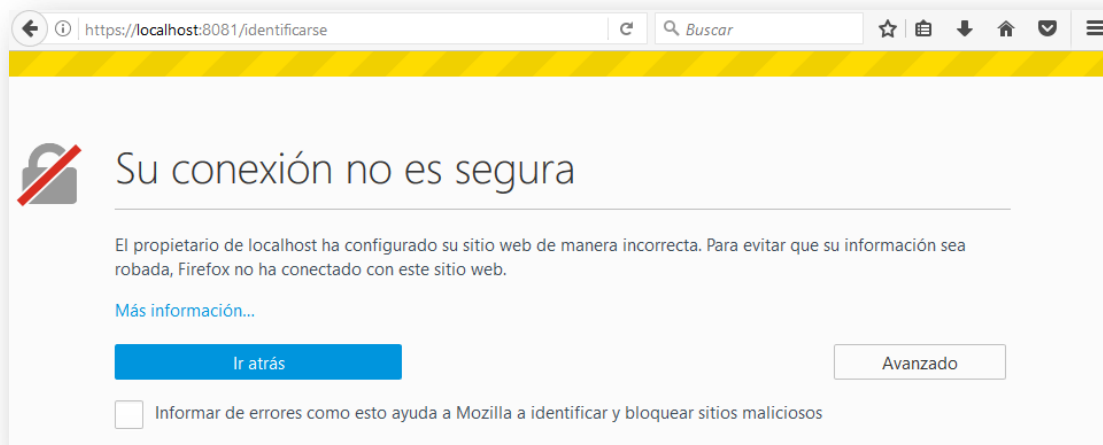
```
https.createServer({  
  key: fs.readFileSync('certificates/alice.key'),  
  cert: fs.readFileSync('certificates/alice.crt')  
}, app).listen(app.get('port'), function() {  
  console.log("Servidor activo");  
});
```

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



Sistemas Distribuidos e Internet

Tema 8 Node JS – Parte 2

