

Integración de aplicaciones

Tema 2. Bases de datos compartidas

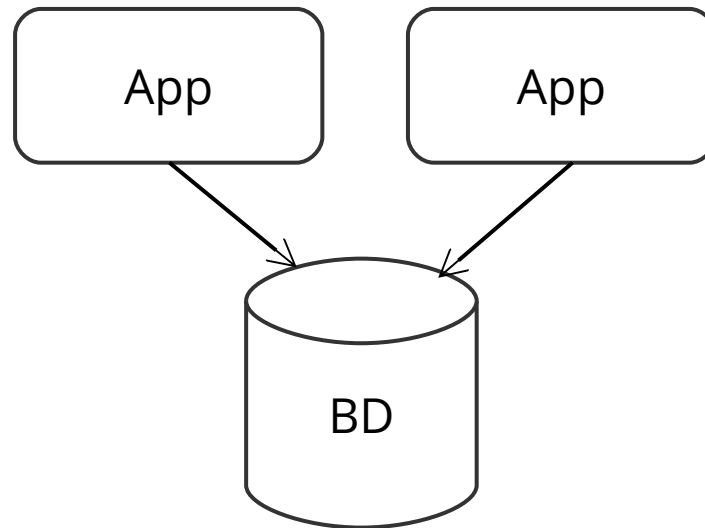
© 2020 Javier Esparza Peidro - jesparza@dsic.upv.es

Contenido

- Introducción
- Consistencia de datos
- Tipos de almacenes
 - MySQL
 - SQLite
 - Redis
 - MongoDB

Introducción

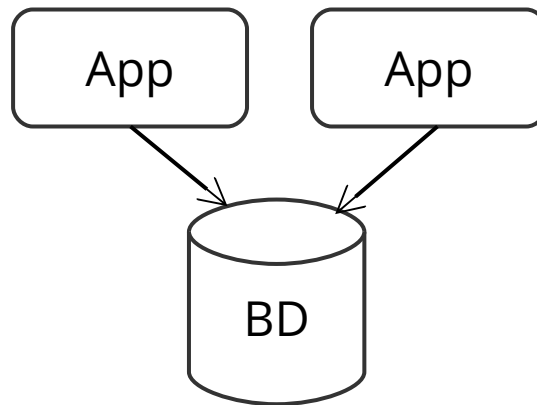
- Integración de aplicaciones = compartir datos
- Podemos diseñar un almacén de datos al que acceden todas las aplicaciones



Introducción

Ventajas

- Es sencillo y directo, no hay intermediarios
- Los datos están centralizados, fuente única de verdad
- Gestión y mantenimiento de datos sencillo



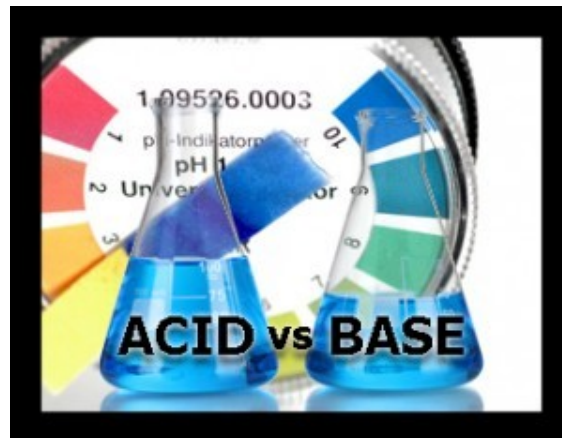
Desventajas

- [illegible]

Consistencia de datos

¿Qué es?

- Son las garantías que un sistema ofrece a la hora de escribir datos de manera persistente
- Generalmente se definen como una serie de reglas bien definidas
- Es fundamental comprenderlas, en especial cuando varias aplicaciones acceden de manera concurrente



Consistencia de datos

Modelo ACID

- Modelo de consistencia **fuerte**, basado en **transacciones**, típico en bases de datos relacionales
- Propiedades:
 - **Atomicidad**: operaciones en una transacción se aplican en bloque, o se ejecutan todas o ninguna
 - **Consistencia**: los datos escritos deben cumplir con todas las restricciones de integridad definidas
 - **Aislamiento**: cambios producidos por una transacción no son visibles para las demás hasta que sea confirmada
 - **Durabilidad**: los datos permanecen, incluso si hay fallos

Consistencia de datos

Modelo BASE

- Modelo de consistencia **débil**, sin transacciones, típicas en almacenes **NoSQL**
- Se prima disponibilidad, escalabilidad y velocidad
- Propiedades:
 - **Basically Available**: operaciones lectura/escritura siempre disponibles, incluso si hay fallos
 - **Soft state**: no hay garantías en cuanto a la consistencia de los datos
 - **Eventually consistent**: en algún momento los datos convergen a estado consistente

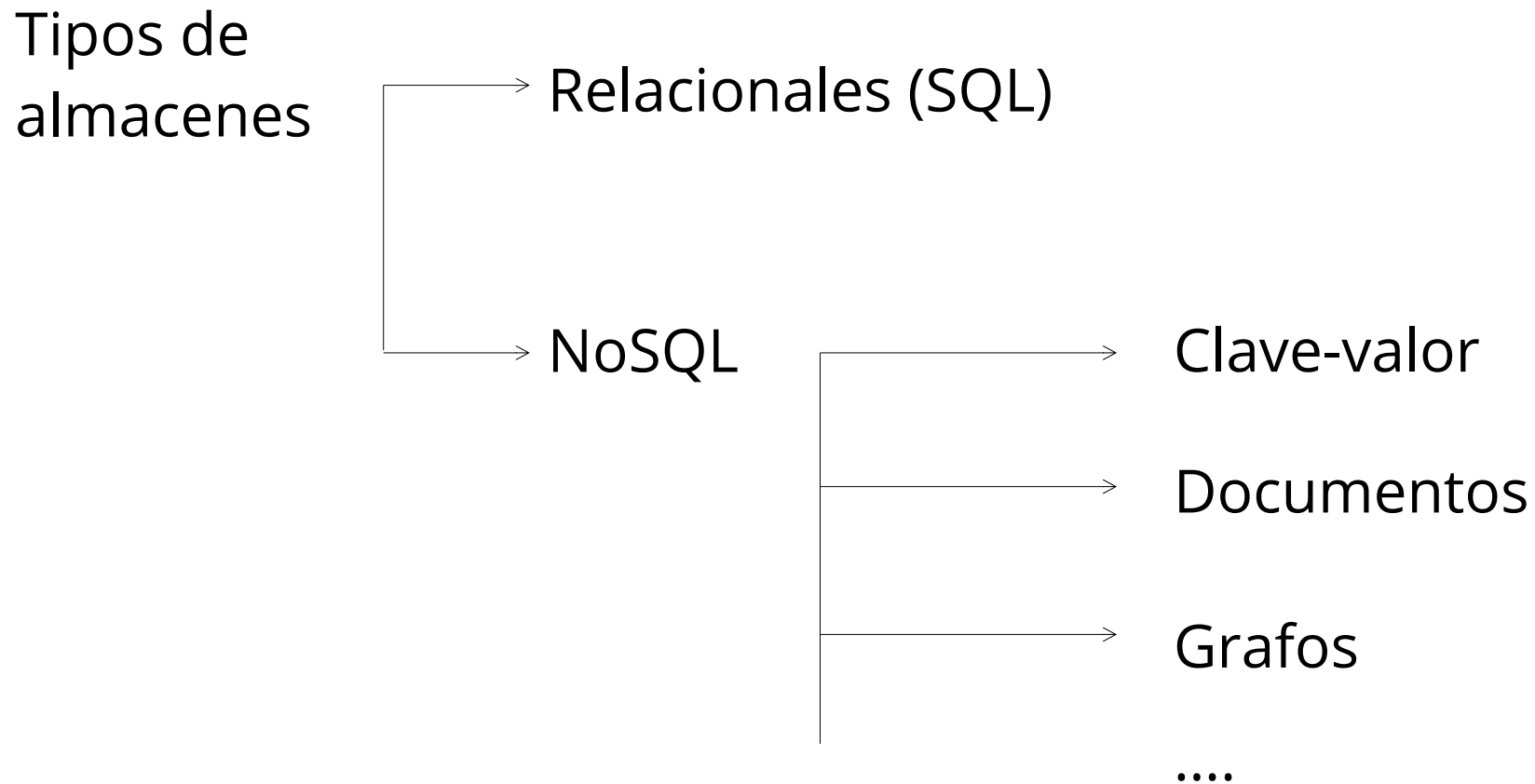
Consistencia de datos

Teorema CAP

- Teorema que demuestra que en un almacén de datos distribuido no es posible proporcionar al mismo tiempo las siguientes garantías:
 - **Consistencia**: cada lectura recibe el valor actual del dato
 - **Disponibilidad**: cada petición recibe siempre respuesta
 - **Tolerancia a particiones**: el sistema funciona incluso en caso de partición de red

Tipos de almacenes

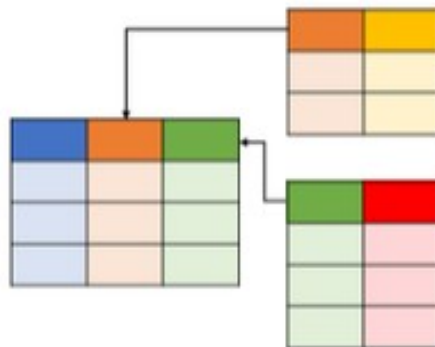
Esquema



Tipos de almacenes

Bases de datos relacionales

- Los datos se almacenan en tablas, y las tablas se relacionan entre sí
- Utilizan el lenguaje SQL
- Las operaciones se incluyen en transacciones
- Suelen implementar las propiedades **ACID**



Tipos de almacenes

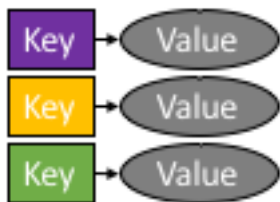
Almacenes NoSQL

- Las bases de datos relacionales (SQL) han dominado la industria durante décadas
- Son maduras, proporcionan integridad de datos, fiabilidad, soporte transaccional (ACID), ...
- Tantas garantías conllevan un coste: no son escalables
- Muchas aplicaciones no requieren tantas garantías, pero necesitan trabajar con un volumen de datos muy grande con mucha velocidad (cloud??)
- Aparecen los sistemas NoSQL (Not Only SQL)

Tipos de almacenes

Almacenes NoSQL

- Almacenes de datos que “relajan” las propiedades ACID en favor de rendimiento, escalabilidad y disponibilidad
- Los datos se organizan en esquemas flexibles (documentos, diccionarios clave-valor, grafos, etc.)
- Hay gran cantidad de motores NoSQL en el mercado: suelen agruparse por el tipo de esquema que ofrecen: clave-valor, documentos, grafos, etc.



Key-Value



Document

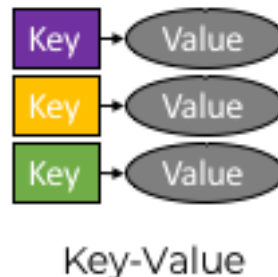


Graph

Tipos de almacenes

Almacenes NoSQL. Almacenes clave-valor

- Es el almacén más simple y genérico
- Colección de pares **clave-valor** indexados por clave
- La clave suele ser string y el valor cualquier cosa
- Recuperación por clave muy eficiente
- Operaciones: put, get, delete
- Se implementan por medio de diccionarios distribuidos y replicados



Tipos de almacenes

Almacenes NoSQL. Almacenes de documentos

- Almacenes clave-valor especializados: valores no son opacos, son **documentos** con estructura interna
- Documento = objeto con propiedades, que pertenecen a distintos tipos de datos
- Esquemas flexibles, anidamiento y referencias
- Disponen de APIs y lenguajes de consulta
- Ofrece garantías de consistencia de datos



Document

Tipos de almacenes

Almacenes NoSQL. Almacenes de grafos

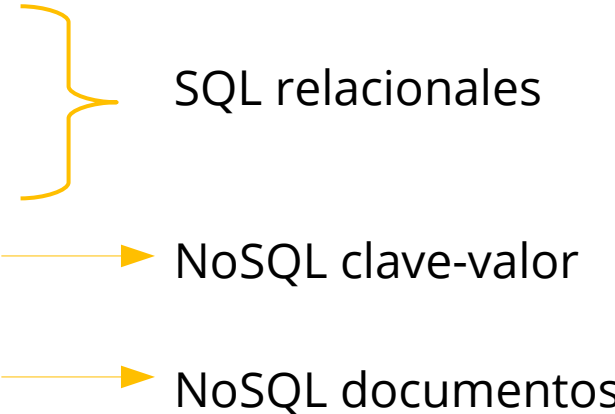
- Extensión de almacenes clave-valor donde además se implementa el concepto de **relación** entre valores
- Se pueden especificar grafos: nodos y aristas (opacos)
- Muy útiles cuando lo que importa es la relación entre entidades
- Optimizados para recuperar de manera eficiente estructuras jerárquicas complejas



Graph

Tipos de almacenes

Hoja de ruta

- Investigar algunos almacenes y ver cómo podemos trabajar con ellos desde Node.js
 - MySQL
 - SQLite
 - Redis
 - MongoDB
- SQL relacionales
- NoSQL clave-valor
- NoSQL documentos
- 

Introducción

- <https://www.mysql.com/>
- Motor de bases de datos relacional clásico
- El servidor se ejecuta en un proceso independiente y sirve las consultas recibidas a través de la red
- Instalaremos la versión MySQL Community Server
- `mysqld` (servidor), `mysql` (CLI-*Command Line Interface*)

```
> mysqld
> mysql -u root -p
mysql> help
mysql> show databases;
mysql> use test;
```

```
mysql> create table xxx(...);
mysql> insert into xxx
      values(...);
mysql> select * from xxx;
mysql> quit
```

Node.js

- Se utiliza un driver, por ejemplo [mysql](#)
 - > `npm install mysql`
- Seguimos siempre los mismos pasos:
 1. Importar paquete `mysql`
 2. Crear conexión contra la base de datos
 3. Ejecutar la consulta/sentencia y procesar resultados
 4. Cerrar conexión

```
const mysql = require("mysql");
let con = mysql.createConnection({host:"",user:"",password:"",database:"",port:..});
con.connect(err => {
  con.query('select * from users', function(err,results) {
    console.log(results); con.end();
  });
});
```

Node.js

1. Importar paquete mysql

```
const mysql = require("mysql");
```

2. Crear conexión contra la base de datos

```
let con = mysql.createConnection({  
  host:"localhost",user:"root",password:"root",  
  database:"test",port:3306 });  
con.connect(err => {  
  if (err) console.log("Error: " + err);  
  else { ... }  
});
```

Node.js

3. Ejecutar la consulta/sentencia y procesar resultados

```
con.query('select * from users',  
    function(err,results) {  
        if (err) console.log("Error: " + err);  
        else console.log(results);  
    }  
);
```

4. Cerrar conexión

```
con.end();
```

Node.js

```
const mysql = require("mysql");
let con = mysql.createConnection({
  host:"localhost",user:"root",password:"root",
  database:"test",port: 3306 });
con.connect(err => {
  if (err) console.log("Error: " + err);
  else {
    console.log("Connected.");
    con.query('select * from users',
      function(err,results) {
        if (err) console.log("Error: " + err);
        else console.log(results);
        con.end();
      });
  }
});
```

Introducción

- <https://www.sqlite.org>
- Pequeño motor SQL compacto y versátil en C
- Incrustado en navegadores, móviles, etc.
- Serverless: no requiere proceso servidor
- Todos los datos se guardan en un fichero
- `sqlite3` (CLI-Command Line Interface)

Node.js

- Se utiliza un driver, por ejemplo [sqlite3](#)
 - > npm install sqlite3
- Seguimos siempre los mismos pasos:
 1. Importar paquete `sqlite3`
 2. Crear conexión contra la base de datos
 3. Ejecutar la consulta/sentencia y procesar resultados
 4. Cerrar conexión

```
const sqlite3 = require('sqlite3').verbose();
var db = new sqlite3.Database(':memory:');
db.each("SELECT rowid AS id, info FROM lorem", function(err, row) {
    console.log(row.id + ": " + row.info);
});
db.close();
```


Node.js

Conexión

- Con `new sqlite3.Database(file, [mode], [callback])`
- Varias opciones: `file` puede apuntar a un fichero o `":memory:"`, varios modos `OPEN_READONLY` | `OPEN_READWRITE` | `OPEN_CREATE`
- `callback` se invoca al finalizar la operación

```
var sqlite3 = require('sqlite3').verbose();  
var db = new sqlite3.Database('midb.db');
```

Node.js

Sentencias sin resultados

- Con `db.run(sql, [param, ...], [callback])`

```
db.run("UPDATE tbl SET name = 'bar' WHERE id = 2");
```

- Parámetros con ?

```
db.run("UPDATE tbl SET name = ? WHERE id = ?", "bar", 2);
```

- `callback(err)`: se invoca al finalizar la sentencia
 - `this` apunta a la sentencia actual: `.lastID`, `.changes`

```
db.run("INSERT INTO t VALUES (2, 'bye')",  
function(err) {  
    if (err) console.log('Error: ' + err);  
    else console.log(this.lastID);  
});
```

Node.js

Sentencias con resultados

- `db.all(sql,[param,...],callback)`: todos los resultados de una
 - `callback(err, rows)`: `rows` es un vector de objetos (filas)

```
db.all("SELECT * FROM t", function(err, rows) {  
  for (var i=0;i<rows.length;i++) console.log(rows[i].titulo);  
});
```

- `db.each(sql, [param,...], callback)`: cada resultado por separado
 - `callback(err, row)`: `row` es el objeto (fila) actual

```
db.each("SELECT * FROM t", function(err, row) {  
  console.log(row.titulo);  
});
```

Node.js

- Por defecto todas las sentencias corren en paralelo (asíncronamente)

```
db.run("CREATE TABLE t(id INT PRIMARY KEY, titulo TEXT)");  
db.run("INSERT INTO t VALUES (2, 'bye')");
```

Fallará ...

- Se pueden serializar (una se ejecuta detrás de otra) con `db#serialize([callback])`

```
db.serialize(function() {  
  db.run("CREATE TABLE t(id INT PRIMARY KEY, titulo TEXT)");  
  db.run("INSERT INTO t VALUES (2, 'bye')");  
});
```

Funcionará ...

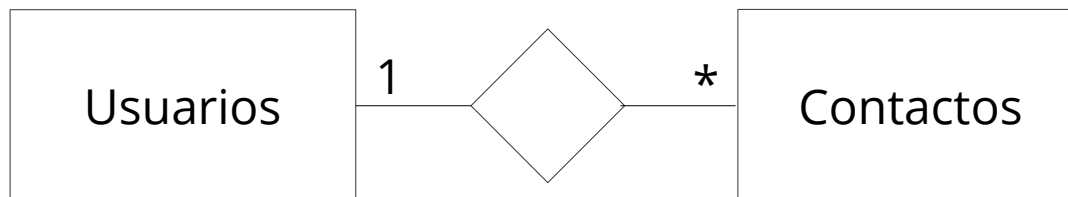


Ejercicio 2

- Diseñar una base de datos SQLite para nuestra aplicación de contactos



- Ampliación multiusuario:
 - Modificar la API para soportar múltiples usuarios



Redis

Introducción

- <https://redis.io/>
- Es un almacén **clave-valor** que almacena toda la información en memoria
- Sirve la información de manera muy eficiente
- Cada cierto tiempo se escriben los cambios en disco (recuperación en caso de fallo)
- Soporta tipos de datos: string, tablas hash, listas, conjuntos, etc.
- Se utiliza como base de datos, cache, cola mensajes

Redis

Instalación

- Oficialmente sólo se soporta en Unix, pero hay [ports a Windows](#)

```
> wget http://download.redis.io/releases/redis-6.0.8.tar.gz
> tar xzf redis-6.0.8.tar.gz
> cd redis-6.0.8
> make
```

- Ejecutar servidor

```
> src/redis-server
```
- Ejecutar cliente

```
> src/redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
redis> keys fo*
["foo"]
```

Redis

Node.js

- Instalar driver [redis](#)

```
> npm install redis
```

- Ejemplo

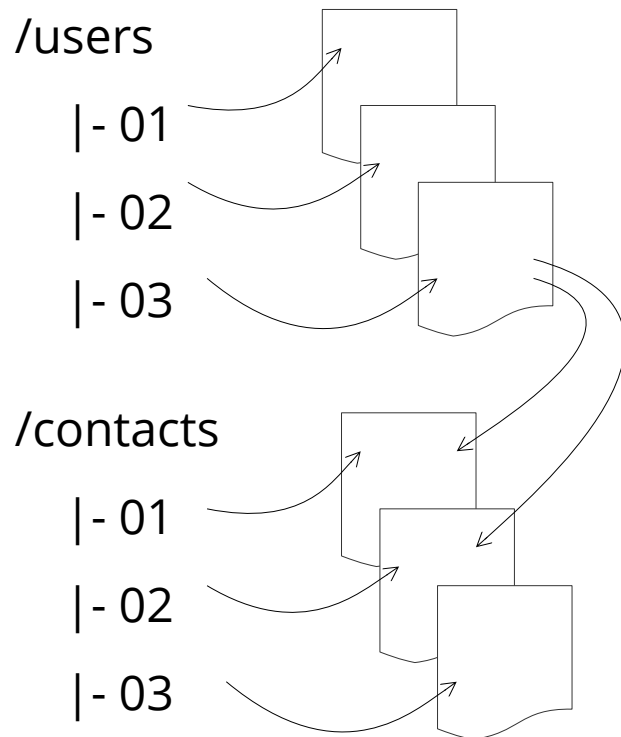
```
const redis = require("redis");  
const client = redis.createClient();  
client.on("error", function(error) {  
    console.error(error);  
});  
client.connect();  
client.set("key", "value").then().catch();  
client.get("key").then().catch();  
client.keys("key").then().catch();
```


Redis

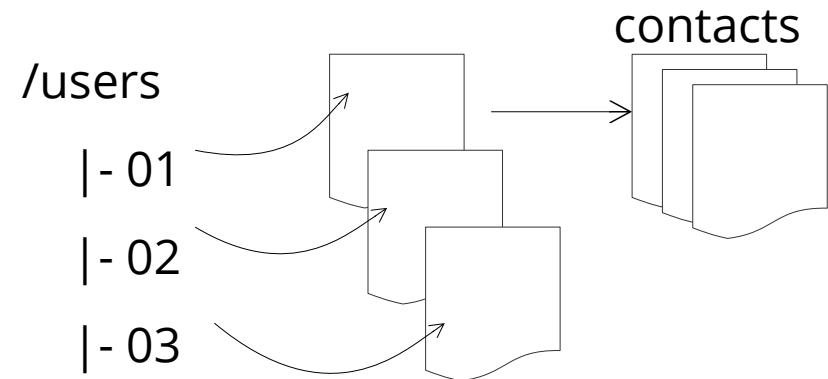


Ejercicio 3

- Portar el modelo de datos de la aplicación de contactos a un diccionario



VS



Introducción

- <https://www.mongodb.com/>
- Repositorio de **documentos** NoSQL open source
- Características clave:
 - Alto rendimiento: índices muy rápidos
 - Alta disponibilidad: con replicación, recuperación automática
 - Escalado automático: utilizando el concepto de **sharding** (distribución horizontal de datos)

Instalación

- Es necesario leer bien cuáles son los requisitos y qué sistemas operativos están soportados
 - En Linux: gestor de paquetes, o bien un tar.gz
 - En Windows: instalador .msi

Configuración

- La configuración de MongoDB reside en `/bin/mongod.cfg` (`/etc/mongod.conf`)
- MongoDB necesita un directorio donde almacenar datos/logs

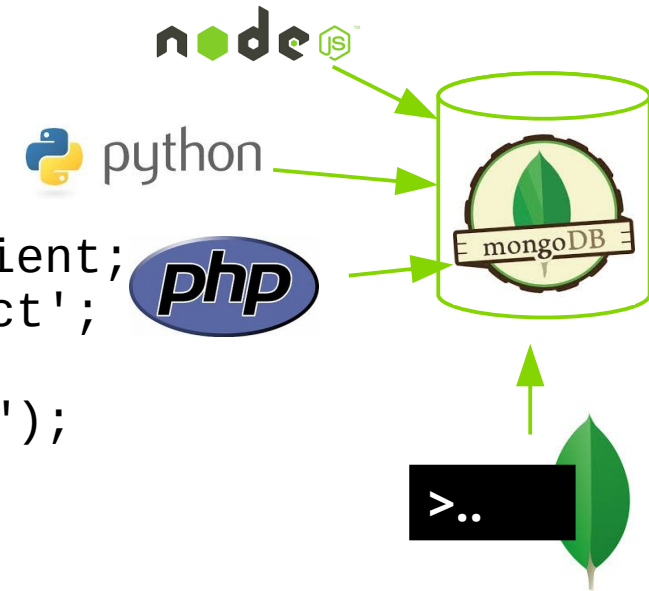
Arrancar el servidor

- Servicio Windows
- Binario: `mongod` `mongod.exe`
> `sudo systemctl start mongod`

Conectar con el servidor

1. A través del driver adecuado

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/myproject';
MongoClient.connect(url, function(err, db) {
  console.log("Connected correctly to server");
  db.close();
});
```



2. A través del cliente MongoDB shell

- Binario: mongosh mongosh.exe
 - > <MONGO_HOME>/bin/mongosh
- Consola interactiva que nos permite interactuar con el servidor

MongoDB shell

- Intérprete interactivo de JavaScript que nos permite interactuar con el servidor
 - > help ————— | Lista comandos disponibles
- mongo vs mongosh
- Permite acceder a MongoDB utilizando un API JS
- No es el mismo API que el driver de Node.js (parecido)
- Muy útil para explorar y administrar el servidor
- Lo usaremos en los próximos ejemplos

Base de datos

- Listar todas las bases de datos
 - > `show dbs;`
- Seleccionar una base de datos para trabajar con ella
 - > `use mibd;`
- La base de datos actual siempre está en la variable **db**
- Una base de datos se crea la primera vez que se inserta en ella
 - > `use nuevabd;` > `db.micoleccion.insert({ ... });`

Documentos

- Un registro en MongoDB es un documento
- Un documento es una estructura de datos compuesta por pares campo-valor
- Muy parecido a un objeto JSON

```
{  
  nombre: "Pepe",  
  edad: 30,  
  estado: "casado",  
  grupos: ["deportes", "noticias"]  
}
```

_____ | campo: valor

Documentos

- Los documentos son almacenados en formato BSON (representación binaria de JSON)
- El valor de un campo puede ser cualquier tipo de datos soportado por BSON, otro documento, un array o un array de documentos

```
{
  nombre: "Pepe",
  direccion : {
    calle : "El Cid",
    cp : "03800",
    coord : [ -73.9557413, 40.7720266 ],
  }
}
```

Documento embebido
(subdocumento)

Colecciones

- Los documentos se guardan en colecciones
- Una colección es el análogo a una tabla en bbdd relacionales
- Una colección no fuerza ninguna estructura (esquema) en sus documentos (pueden ser diferentes)
- Generalmente todos los documentos en una colección poseen una estructura similar, y comparten índices



Colecciones

- Mostrar las colecciones de la base de datos actual
 - > `show collections`
- Acceder a una colección en la base de datos actual
 - > `db.usuarios`

Colecciones > Insertar documentos

- Usamos `db.collection.insertOne(doc, [opts])`
 - doc: documento a insertar
 - opts: opcional. Opciones adicionales

```
> db.usuarios.insertOne({  
  "nombre": "Pepe", "edad": "30", "estado": "casado"  
});
```
- Si la colección no existe se creará automáticamente
- Devuelve info sobre el documento insertado (insertedId)
- `db.collection.insertMany()`

Colecciones > Insertar documentos

- Los documentos almacenados en una colección deben tener un identificador único `_id` (`ObjectId`) que es la clave primaria en dicha colección
- Si no los genera el cliente, lo generará automáticamente el servidor

```
> db.usuarios.insertOne({  
  _id: ObjectId(),  
  "nombre": "Pepe", "edad": "30", "estado": "casado"  
});
```

Colecciones > Buscar documentos

- Usamos `db.collection.find([query], [projection])`
 - query: opcional. Criterios de búsqueda
 - projection: opcional. Limita los campos a recuperar
- Obtenemos un cursor con los resultados
 - > `var cursor = db.usuarios.find();`
- En MongoDB shell se iteran y muestran automáticamente los primeros 20 documentos. Para mostrar más usar **it**
 - > `db.usuarios.find();`
 - > `it`

Colecciones > Buscar documentos



Cursor

- Apunta a los resultados de una query
- [cursor.next\(\)](#) [cursor.hasNext\(\)](#)
- Otros [métodos](#)

```
> var cursor = db.usuarios.find();  
> while (cursor.hasNext()) {  
    print(cursor.next());  
}
```

Colecciones > Buscar documentos



Query

- Especificar condiciones de igualdad

```
{ <campo1>: <valor1>, <campo2>: <valor2>, ... }
```

```
> db.usuarios.find({_id: 1});
```

```
> db.usuarios.find({nombre: 'XXX', apellidos: 'YYY'});
```

- Si <campo> está dentro de un documento embebido entonces se especifica con ., y con comillas

```
> db.usuarios.find({nombre: 'XXX', 'direccion.ciudad':  
  'Alcoy'});
```


Colecciones > Buscar documentos



Query

- Especificar condiciones con operadores de consulta
 - > `db.usuarios.find({edad: {$gt: 50} });`
 - > `db.productos.find({color: {$in: ['rojo', 'verde']}});`
- `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$in`, `$nin`
- `$or`, `$and`, `$not`, `$nor`
- `$exists`, `$type`, `$mod`, `$regex`, `$text`, `$where`, `$all`, `$elementMatch`, `$size`, `$slice`, ...

Colecciones > Modificar documentos

- Usamos `db.collection.updateOne(query, update, [opts])`
 - query: determina los documentos a modificar; usamos las mismas condiciones que con las búsquedas
 - update: especifica las modificaciones a aplicar
 - opts: opcional. Opciones adicionales

```
> db.usuarios.updateOne({ _id: "0001"}, query  
  { $set: { "name", "Juan" } } modificaciones  
);
```
- Devuelve (.matchedCount, .modifiedCount, upsertedId,...)
- `db.collection.updateMany()`

Colecciones > Modificar documentos

Modificaciones

- Se especifican con operadores de modificación : \$inc, \$mul, \$rename, \$setOrInsert, \$set, \$unset, \$min, \$max, \$currentDate, \$addToSet, \$pop, \$pull, \$each, \$slice, \$sort, ...

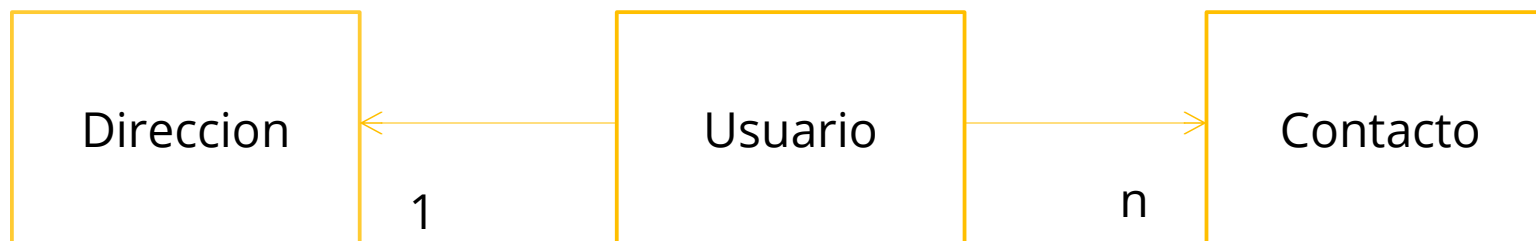
```
> db.usuarios.updateOne( { name: "Pepe"},  
  { $set: { "name", "Juan" }, $inc: { "edad": 1 } }  
);
```
- Algunos operadores crean el campo si no existe
- Las modificaciones son **atómicas** en un documento

Colecciones > Eliminar documentos

- Usamos [db.collection.deleteOne\(query, \[opts\]\)](#)
 - query: determina los documentos a eliminar; usamos las mismas condiciones que con las búsquedas
 - opts: opcional. Opciones adicionales
 - > `db.usuarios.deleteOne({ name: "Pepe"});`
- Devuelve info (.deletedCount, ...)
- [db.collection.deleteMany\(\)](#)

Modelado de datos

- Cuando diseñamos un modelo de datos, definimos entidades y relaciones entre éstas



- En MongoDB las entidades se modelan con colecciones
- Podemos modelar las relaciones de dos maneras: documentos embebidos vs referencias

Modelado de datos > Documentos embebidos

- Un documento contiene otros/s documento/s

Usuario

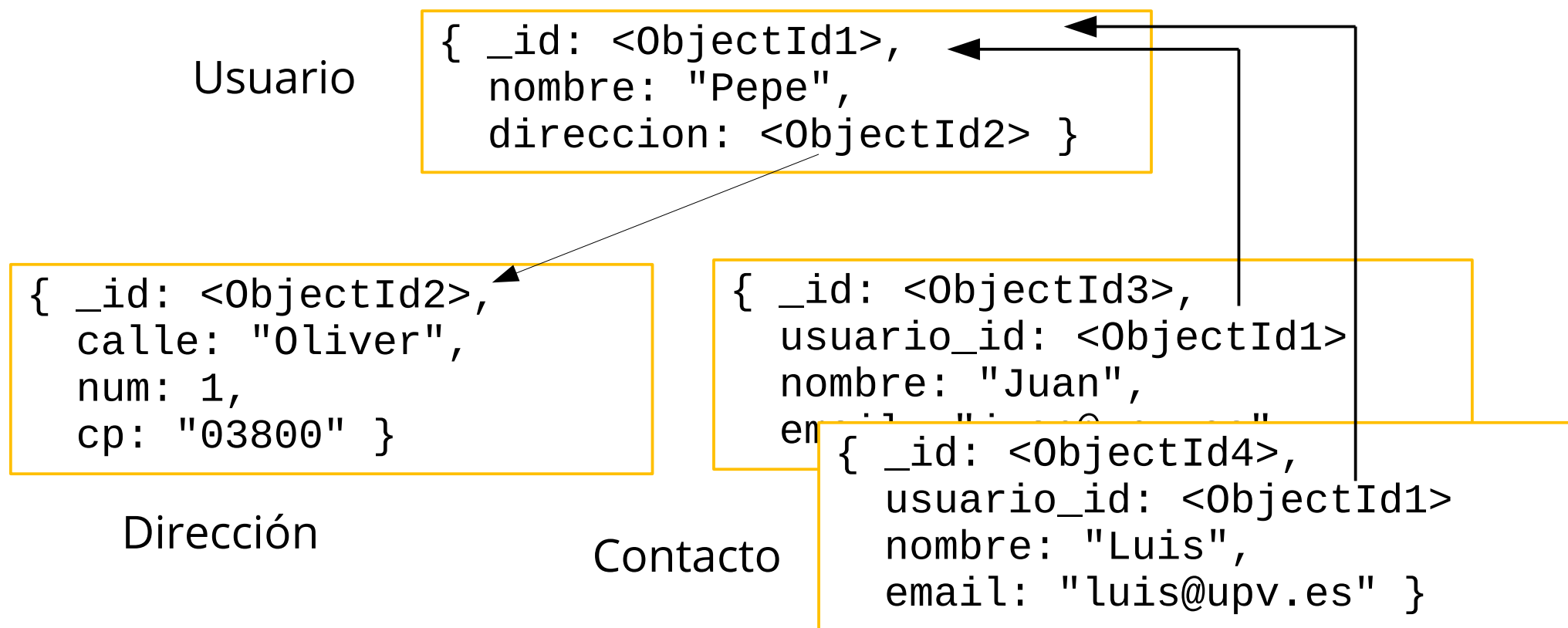
```
{ _id: <ObjectId1>,
  nombre: "Pepe",
  direccion: {
    calle: "Oliver",
    num: 1,
    cp: "03800" },
  contactos: [
    {nombre: "Juan", email: "juan@upv.es" },
    {nombre: "Luis", email: "luis@upv.es" } ]
}
```

} Dirección

} Contacto

Modelado de datos > Referencias

- Un documento apunta a otro/s documento/s, típicamente almacenando su ObjectId



Modelado de datos > Comparativa

- Documentos embebidos
 - Requiere menos consultas
 - Modela muy bien la relación “contiene”
 - Podría necesitar replicar información
- Referencias
 - Más flexibles que los documentos embebidos
 - Para representar relaciones complejas (n a n)
 - Cuando un documento es compartido por varios
 - Con documentos de gran volumen (límites BSON)

Node.js

- Hasta ahora hemos accedido a MongoDB usando el MongoDB shell
- Para acceder a MongoDB desde Node.js es necesario utilizar el driver adecuado
- Similar a MongoDB shell (JS) pero distinta API
- En Node.js todas las llamadas son asíncronas (callbacks v4 ó promesas v5+)
- Instalación
 - > `npm install mongodb[@4.17]`

Node.js

Conexión

- La creamos usando `new MongoClient(url)`
- url: `'mongodb://localhost:27017'`
- Devuelve un cliente

```
const MongoClient = require('mongodb').MongoClient;  
const client = new MongoClient('mongodb://localhost:27017');  
client.connect()  
...  
client.close();
```

Node.js

Base de datos

- Accedemos a la base de datos con `client.db(name, [opts])`
`var db = client.db('test');`
- Devuelve un objeto de tipo `Db`
- Nos permite acceder a las colecciones usando `db.collection(name, [opts])`
`var col = db.collection('usuarios');`
- Devuelve un objeto de tipo `Collection`

Node.js

Colección > Buscar documentos

- Collection.findOne(query, [opts] [cb])
- Collection.find(query, [opts])
 - query: la query, con la sintaxis que ya conocemos
 - [opts]: opcional. Muchas opciones: limit, sort, fields, skip, ...
 - cb(err, result) o promesa: sólo para findOne()
 - Devuelve un cursor (findOne() no)

```
var cursor = db.collection('usuarios').find(  
  {edad: {$gt: 50}},  
  {skip:1, limit:1, projection:{nombre:1}});
```

Node.js

Colección > Buscar documentos

- Cursor
 - Objeto de tipo Cursor
 - .next([cb]): siguiente resultado, o null
 - .toArray([cb]), .forEach(iterator,[cb]), .filter(filter), etc.

```
db.collection('usuarios').find({}).toArray(  
  function(err, docs) {  
    for (var i = 0; i < docs.length; i++)  
      console.log(docs[i].nombre);  
  }  
);
```

Node.js

Colección > Insertar documentos

- Collection.insertOne(doc, [opts], cb)
- Collection.insertMany(docs, [opts], cb)
 - doc, docs: documento/s a insertar
 - [opts]: opcional. Opciones varias
 - cb(err, res) o promesa

```
db.collection('usuarios').insertOne(  
  {nombre: 'Pepe', edad: 50},  
  function(err, res) { ... }  
);
```

Node.js

Colección > Actualizar documentos

- Collection.updateOne(query, update, [opts], cb)
- Collection.updateMany(query, update, [opts], cb)
 - query: selecciona los elementos a modificar
 - update: operaciones de modificación
 - [opts]: opcional. Opciones adicionales: upsert

```
db.collection('usuarios').updateMany(  
  {nombre: 'Pepe'},  
  {$set: {edad: 50}},  
  function(err, res) { ... });
```

Node.js

Colección > Eliminar documentos

- [Collection.deleteOne\(query, \[opts\], cb\)](#)
- [Collection.deleteMany\(query, \[opts\], db\)](#)

```
db.collection('usuarios').deleteMany(  
  {nombre: 'Pepe'},  
  function(err, res) { ...}  
);
```


MongoDB



Ejercicio 4

- Migrar el modelo de datos a MongoDB

