UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Integración de aplicaciones

Laboratorio 1

Contenido

1. Introducción

Este laboratorio persigue los siguientes objetivos:

1. Introducir al alumno a la tecnología base con la que se trabajará a lo largo del curso: Node.js.
2. Implementar una arquitectura de aplicación desacoplada que servirá de base para el resto de laboratorios.
3. Poner en práctica los conceptos de integración por almacén de datos compartido, en concreto utilizando el almacén de documentos MongoDB.

Para ello, se propone el desarrollo de una aplicación de envío de mensajes. Se denominará Twitter Little. Se pondrá especial énfasis en la arquitectura de la aplicación, para que resulte reutilizable en sucesivos laboratorios.

En la sección 2 se presenta la funcionalidad de la aplicación. En la sección 3 se propone la arquitectura a utilizar. En el apartado 4 se propone el modelo de datos a utilizar. En el apartado 5 se propone la API a implementar y finalmente en el apartado 6 se propone un esqueleto con el que implementar la interacción con el usuario.

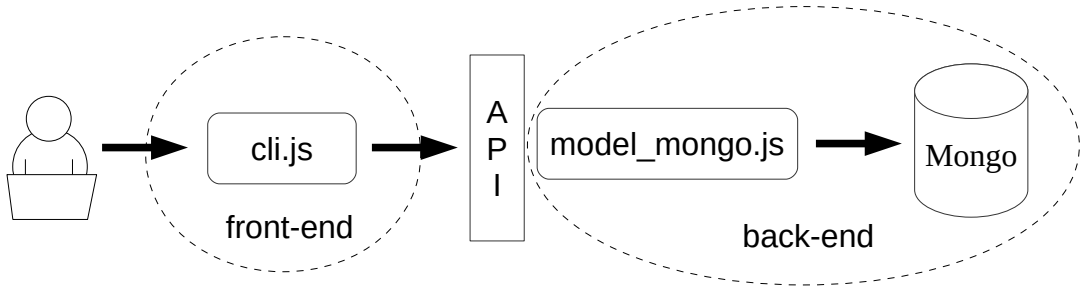
2. Twitter Lite

Como su nombre sugiere, Twitter Lite representa una versión limitada de Twitter. A grandes rasgos, el usuario podrá ejercitar las siguientes funcionalidades:

- Registrar nuevos usuarios.
- Autenticarse en la aplicación.
- Enviar mensajes (tweets).
- Buscar otros usuarios.
- (Seguir / Dejar de seguir) a otro usuario.
- Leer los mensajes de los usuarios seguidos.
- Registrar likes (me gusta) o dislikes (no me gusta) en los mensajes (tweets) enviados por otros usuarios.
- Explorar los mensajes enviados por otros usuarios, a los que no se sigue.

3. Arquitectura de la aplicación

La aplicación presentará la arquitectura mostrada en la siguiente figura:



Como se puede observar, la aplicación incluirá dos módulos:

- *cli.js*: incluye la interacción con el usuario. Se trata de una herramienta CLI (*Commando Line Interface*) que permitirá al usuario interaccionar con el sistema, en línea de comandos. Para ello, accederá a la API publicada por el módulo *model_mongo.js*.

- *model_mongo.js*: publica una API con todas las operaciones de la aplicación e implementa el modelo de datos. En este laboratorio, la información se almacenará en MongoDB. En

sucesivos laboratorios iremos modificando este fichero para que la información sea obtenida de diversas fuentes. El objetivo es que la API publicada aquí sufra el menor número de cambios, para que el resto de la aplicación (en esencia *cli.js*) no se resienta.

Con esta arquitectura, se obtiene una solución con acoplamiento muy débil, ya que la API hace de interfaz entre el front-end y el back-end. Si la API se mantiene estable, resulta muy sencillo reemplazar el back-end utilizando distintas tecnologías, sin necesidad de modificar una línea del front-end.

A continuación se procederá a desarrollar la aplicación Node.js. Para ello, primero se creará el directorio raíz del proyecto, por ejemplo */twitter,* y se ejecutará el siguiente comando:

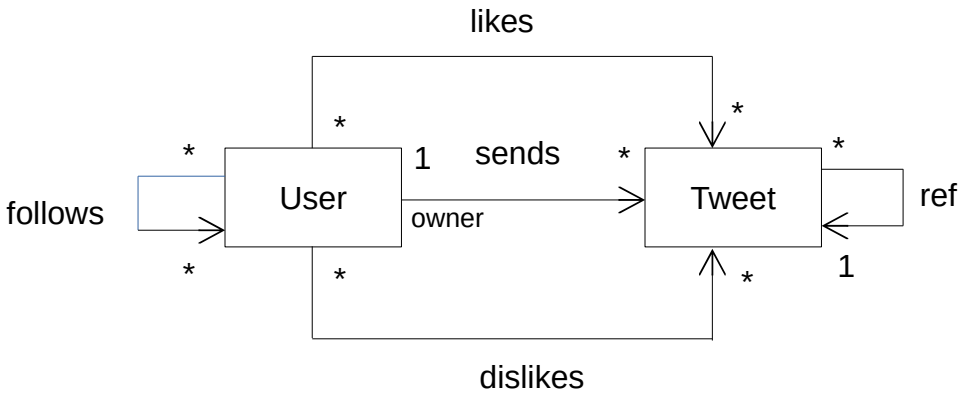
```
> npm init
```

Se plantea la siguiente estructura de proyecto inicial:

```
twitter
  |_ cli.js
  |_ model_mongo.js
  |_ package.json
```

4. Modelo de datos

Para implementar todas las funcionalidades descritas se propone el siguiente modelo de datos.

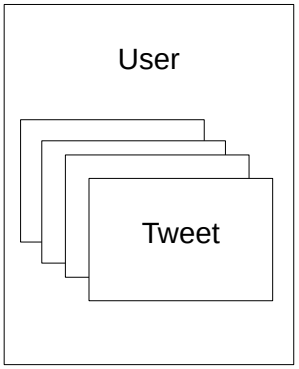


El diagrama es fácil de interpretar. Un usuario sigue a otros (asociación *follows*). Además, un usuario envía tweets o retweets. En este último caso, el retweet referencia al tweet original (asociación *ref*). Por último, un usuario puede registrar *likes* o *dislikes* sobre los tweets enviados por otros usuarios.

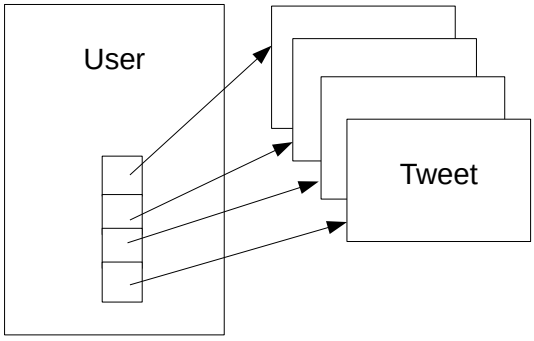
Un usuario podría incluir al menos las siguientes propiedades: *id*, *name*, *surname*, *email*, *password*, *nick*. Por su parte, un tweet podría incluir como mínimo las siguientes propiedades: *id*, *date*, *content*.

Como ya hemos visto en clase, cuando se trabaja con un almacén de documentos, las referencias entre entidades se pueden modelar bien como documentos embebidos, bien como referencias entre documentos. En nuestro caso, podemos optar por uno de los dos modelos básicos siguientes:

Documentos embebidos



Referencias



En el primer modelo un usuario contiene todos sus tweets mientras que en el segundo modelo un usuario referencia sus tweets. Si pensamos en el primer modelo, encontraremos dificultades para modelar retweets, ya que estos deben de mantener referencias a un tweet, y éste podría pertenecer a cualquier usuario, por lo que deberíamos de buscarlo en el interior de todos los usuarios.

Resulta por tanto más conveniente mantener dos colecciones de documentos: Users y Tweets. De este modo, los tweets tienen existencia independiente y pueden referenciarse entre sí sin inconveniente alguno.

Además, será necesario decidir cómo modelar las relaciones 1-n y n-n identificadas en el modelo original (*follows*, *likes*, *dislikes*). Para el caso de la relación *follows*, una posible opción es que el usuario contenga un vector *following* de ObjectId que determina los usuarios a los que sigue uno dado. Del mismo modo es posible modelar las otras relaciones *likes* o *dislikes*.

5. La API

A continuación se presenta la API a implementar. Primero se presentan las operaciones relacionadas con usuarios y después las operaciones relacionadas con los tweets.

Usuarios		
Función	**Parámetros**	**Descripción**
addUser(user,cb)	-user: el usuario -cb(err,user): el callback	Añade un nuevo usuario. El callback recibe el usuario creado.
login(email,passwd,cb)	-email: el email -passwd: el password -cb(err, token, usr)	Autentica a un usuario. El callback recibe un token y el usuario autenticado. El token identifica al usuario y permitirá el acceso al resto de operaciones.
updateUser(token,user,cb)	-token: obtenido con login -user: los datos a actualizar -cb(err,user)	Actualiza el usuario autenticado con los datos suministrados. El callback recibe el usuario actualizado.
listUsers(token,opts,cb)	-token: obtenido con login -opts.query: contiene la consulta -opts.ini: índice del primer resultado -opts.count: número máximo de resultados	Lista los usuarios, con diversas opciones. El callback recibe un vector de usuarios.

	-opts.sort: ordenar resultados por (+|-)campo -cb(err, usrs)	
listFollowing(token,opts, cb)	-token: obtenido con login -opts.query: contiene la consulta -opts.ini: índice del primer resultado -opts.count: número máximo de resultados -opts.sort: ordenar resultados por (+|-)campo -cb(err, usrs)	Lista los usuarios a los que sigue el usuario especificado por token, con diversas opciones. El callback recibe un vector de usuarios.
listFollowers(token,opts, cb)	-token: obtenido con login -opts.query: contiene la consulta -opts.ini: índice del primer resultado -opts.count: número máximo de resultados -opts.sort: ordenar resultados por (+|-)campo -cb(err, usrs)	Lista los usuarios que siguen al usuario especificado por token, con diversas opciones. El callback recibe un vector de usuarios.
follow(token,userId,cb)	-token: obtenido con login -userId: el usuario a seguir -cb(err)	El usuario solicita seguir a otro.
unfollow(token,userId,cb)	-token: obtenido con login -userId: el usuario a dejar de seguir -cb(err)	El usuario solicita dejar de seguir a otro.
Tweets		
Función	**Parámetros**	**Descripción**
addTweet(token,content, cb)	-token: obtenido con login -content: el mensaje cb(err, tweet)	Añade un nuevo tweet. El callback recibe el tweet creado.
addRetweet(token,tweetId, cb)	-token: obtenido con login -tweetId: el tweet retweeteado cb(err,retweet)	El usuario autenticado reteetea otro tweet El callback recibe el retweet creado.
listTweets(token,opts,cb)	-token: obtenido con login -opts.query: contiene la consulta -opts.ini: índice del primer resultado -opts.count: número máximo de resultados -opts.sort: ordenar resultados por (+|-)campo -cb(err,tweets)	El usuario autenticado lista todos los tweets, con diversas opciones. El callback recibe un vector con los tweets (incluyendo tweets y retweets).
like(token,tweetId,cb)	-token: obtenido con login -tweetId: el id del tweet cb(err)	El usuario indica que le gusta un tweet
dislike(token,tweetId,cb)	-token: obtenido con login -tweetId: el id del tweet cb(err)	El usuario indica que no le gusta un tweet

Es interesante efectuar los siguientes comentarios sobre la API:

- Autenticación: se propone un mecanismo de autenticación basado en tokens. Para ello, el usuario suministra sus credenciales (email y password) y si la autenticación tiene éxito el sistema devuelve un *token*. Un *token* es una “llave” que permite al usuario ejecutar las operaciones “sensibles” del sistema (todas menos la creación de usuario y autenticación).

- Las consultas: se propone al usuario implementar un mecanismo de consultas similar al de MongoDB. De este modo, una consulta que pretende obtener los usuarios “Pepe” podría efectuarse así:

  ```
  listUsers(token, {query: {name: “Pepe”}}, cb)
  ```

  Del mismo modo, se pueden efectuar consultas más complejas utilizando *operadores de consulta.* Un ejemplo sería recuperar los tweets comprendidos en cierto rango de tiempo:

  ```
  listTweets(token,{query:{date:{$gte:1633284222739,$lt:1633284222739}}},cb)
  ```

A continuación, implementaremos la librería *model_mongo.js*, En esta librería iremos implementando las distintas operaciones de la API. Se muestran algunos ejemplos. Comenzamos por *addUser().*

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;
const url = 'mongodb://localhost:27017';

function addUser(user, cb) {
  if (!user.name) cb(new Error('Property name missing'));
  else if (!user.surname) cb(new Error('Property surname missing'));
  else if (!user.email) cb(new Error('Property email missing'));
  else if (!user.nick) cb(new Error('Property nick missing'));
  else if (!user.password) cb(new Error('Property password missing'));
  else {
    MongoClient.connect(url).then((client) => {

      // create new callback for closing connection
      _cb = function (err, res) {
        client.close();
        cb(err, res);
      }

      let db = client.db('twitter_lite');
      let users = db.collection('users');
      users.findOne({ $or:[{ email: user.email },{ nick: user.nick }] }).then(
        (_user) => {
          if (_user) _cb(new Error('User already exists'));
          else {
            user.following = [];
            users.insertOne(user).then(result => {
              _cb(null, {
                id: result.insertedId.toHexString(), name: user.name,
                surname: user.name, email: user.email, nick: user.nick
              });
            }).catch(err => {
              _cb(err)
            });
          }
        }).catch(err => {
          _cb(err)
        });

      }).catch(err => {
```

```
      cb(err);
    });
  }
}
```

Como se puede observar, una vez creada la conexión contra MongoDB, se crea una función *_cb* que envuelve el callback *cb*, y su propósito es cerrar la conexión abierta. De este modo, antes de finalizar la operación, será necesario invocar *_cb*, que cerrará la conexión e invocará al callback real.

Por otro lado, es importante resaltar que los identificadores almacenados en MongoDB son instancias de la clase *mongodb.ObjectId*. Sin embargo, para la API los identificadores serán *String*, ya que no queremos acoplarla a un almacén de datos en concreto. Será por tanto necesario efectuar una conversión entre los identificadores almacenados internamente en MongoDB y los identificadores que mostraremos a la interfaz de usuario. Para transformar un objeto *mongodb.ObjectId* a *String*, basta con invocar el método *mongodb.ObjectId.toHexString()*. Para efectuar la operación inversa podemos utilizar el contructor *mongodb.ObjectId()*, pasándole como parámetro el *String* correspondiente.

A continuación implementaremos la operación login. Se puede observar que el token que se genera cuando la autenticación tiene éxito es simplemenet el id del usuario. En una implementación sería este token debería de cifrarse y contener al menos el id del usuario y el instante de tiempo en que se concedió dicho token.

```
function login(email, password, cb) {
  MongoClient.connect(url).then(client => {

    // create new callback for closing connection
    _cb = function (err, res, res2) {
      client.close();
      cb(err, res, res2);
    }

    let db = client.db('twitter_lite');
    let col = db.collection('users');
    col.findOne({ email: email, password: password }).then(_user => {
      if (!_user) _cb(new Error('Wrong authentication'));
      else {
      _cb(null, _user._id.toHexString(), {
        id: _user._id.toHexString(), name: _user.name, surname: _user.surname,
        email: _user.email, nick: _user.nick });
    }}).catch(err => {
      _cb(err)
    });
  }).catch(err => {
    cb(err);
  });
}
```

Mostramos también como ejemplo la operación *listUsers()*, para ilustrar de qué manera podemos adaptar las consultas a la nueva sintaxis impuesta por MongoDB.

```
function listUsers(token, opts, cb) {
  MongoClient.connect(url).then(client => {

    // create new callback for closing connection
    _cb = function (err, res) {
      client.close();
      cb(err, res);
```

```
    }

    let db = client.db('twitter_lite');
    let users = db.collection('users');
    users.findOne({ _id: new mongodb.ObjectId(token) }).then(_user => {
      if (!_user) _cb(new Error('Wrong token'));
      else {
        // adapt query
        let _query = opts.query || {};
        for (let key in _query) {
          if (Array.isArray(_query[key])) _query[key] = { $in: _query[key] };
        }
        // adapt options
        let _opts = {};
        if (opts.ini) _opts.skip = opts.ini;
        if (opts.count) _opts.limit = opts.count;
        if (opts.sort) _opts.sort = [[opts.sort.slice(1),
          (opts.sort.charAt(0) == '+' ? 1 : -1)]];

        users.find(_query, _opts).toArray().then(_results => {
          let results = _results.map((user) => {
            return {
              id: user._id.toHexString(), name: user.name,
              surname: user.surname, email: user.email, nick: user.nick
            };
          });
          _cb(null, results);
        }).catch(err => {
          _cb(err)
        });
      }
    }).catch(err => {
      _cb(err)
    });

  }).catch(err => {
    cb(err);
  });
}
```

Finalmente, recordar exportar las operaciones en el módulo.

```
module.exports = {
    addUser,
    login,
    listUsers
}
```

6. El CLI

La interacción con el usuario se efectuará a través de una herramienta CLI (*Command Line Interface*), que se incluirá en el módulo *cli.js*. Para simplificar su implementación se recomienda hacer uso de los siguientes componentes adicionales:

- El módulo core *readline,* que permite leer de la entrada estándar de una manera sencilla.
- El paquete *minimist*, que simplifica el parsing de las opciones de entrada.

```
> npm install minimist
```

A continuación se plantea un esqueleto de código que puede resultar de utilidad para la implementación del módulo *cli.js*.

```
const readline = require("readline");
const minimist = require("minimist");
const model = require("./model_mongo");

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.setPrompt("cli > ");
rl.prompt();
rl.on("line", line => {
    let args = minimist(fields = line.split(" "));
    menu(args, () => {
        rl.prompt();
    });
});

function menu(args, cb) {
    if (!args._.length || args._[0] == "") cb();
    else {
        switch (args._[0]) {
           case "help": ...
           case "exit": …
           case "login": ...
           case "addUser": ...
           case "updateUser": …
            …
        }
    }
}
```