UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Integración de aplicaciones

Laboratorio 4

Contenido

1. Introducción

En este laboratorio experimentaremos con el concepto de microservicio y pondremos en práctica nuestros conocimientos sobre contenedores, Docker y Kubernetes.

Para ello, fragmentaremos el servicio RESTful desarrollado en el laboratorio 2 (ó 3) en dos microservicios, cada uno responsable de un único asunto. A continuación los microservicios serán contenerizados con Docker y finalmente se desplegará la aplicación completa utilizando Kubernetes.

2. Microservicios

Si pensamos en ello, nuestra aplicación de tweets posee diversas responsabilidades:

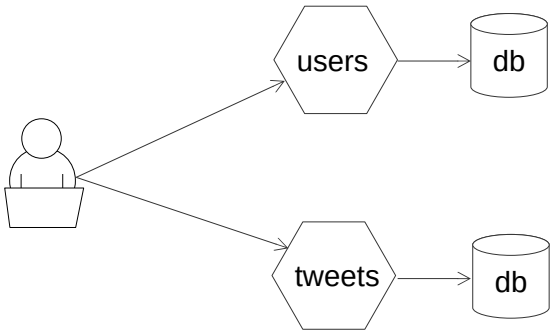
- Gestión de usuarios
- Autenticación de usuarios
- Gestión de seguidores
- Gestión de tweets
- Gestión de retweets
- Gestión de likes y dislikes

Siguiendo la filosofoía orientada a microservicios, podríamos fragmentarla en tantos microservicios como responsabilidades posee. Sin embargo, algunas de estas reponsabilidades están fuertemente relacionadas entre sí y es posible que no encontráramos demasiado beneficio en hacerlo, incurriendo además en una mayor complejidad a la hora de coordinar todos los microservicios.

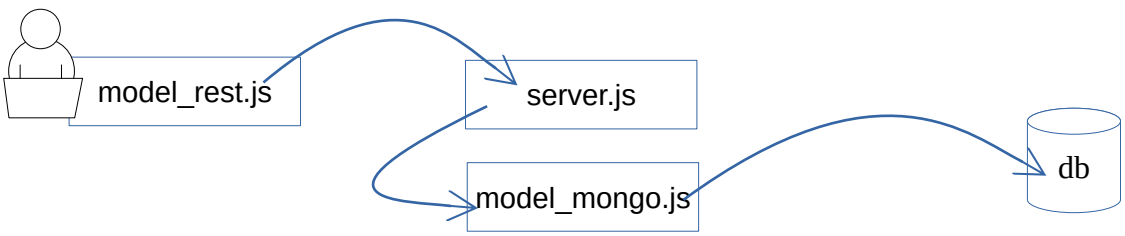
Para simplificar, identificaremos dos microservicios esenciales:

- Microservicio *users*: se encargará de la gestión de usuarios, incluyendo seguidores y autenticación.
- Microservicio *tweets*: se encargará de la gestión de tweets, retweets, likes y dislikes.

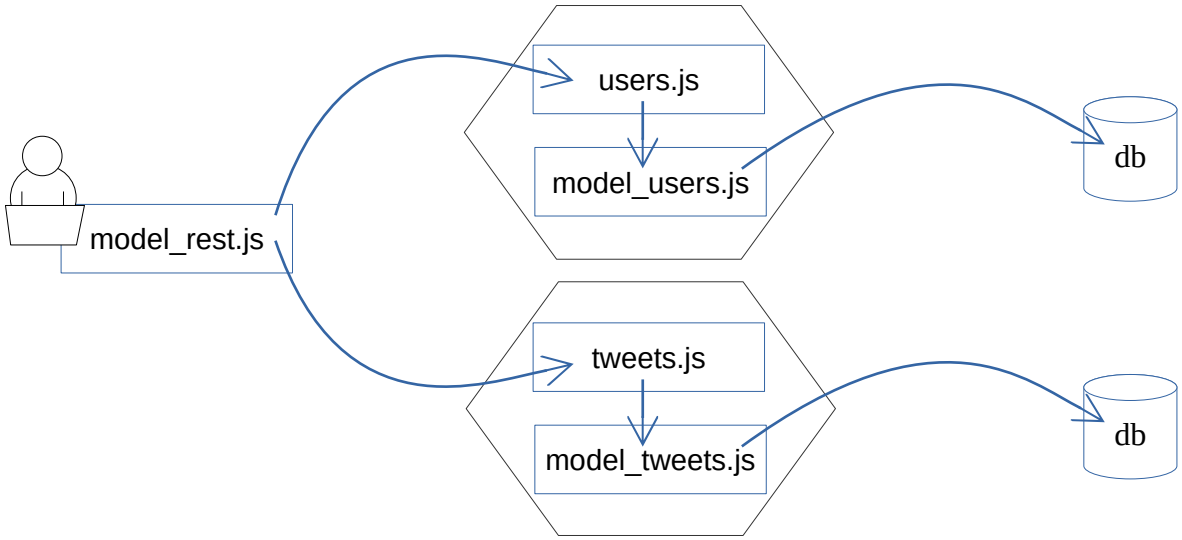
Siguiendo los principios de diseño de las arquitecturas orientadas a microservicios, cada microservicio será autónomo, publicará su propia API y gestionará sus propios datos. La siguiente figura ilustra la nueva arquitectura de nuestra aplicación. Como se puede observar, cada microservicio ataca a su propio almacén de datos.



En el laboratorio 2 se obtuvo una arquitectura similar a la siguiente:



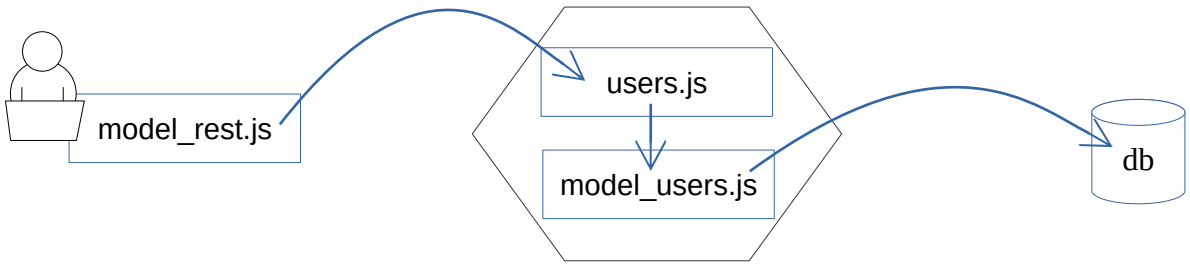
Para crear los microservicios *users* y *tweets* aprovecharemos gran parte del código incluído en *server,js* y *model_mongo.js,* pero cada uno poseerá sus módulos independientes. La arquitectura resultante será similar a lo siguiente:



En las siguientes secciones proporcionamos una guía acerca de cómo se pueden diseñar los nuevos microservicios *users* y *tweets*.

2.1 Microservicio users

El objetivo de eta sección es crear el microservicio *users*, tal y como se describe en la siguiente figura:



Para ello, crearemos dos nuevos módulos *users.js* y *model_users.js*, que incluirán la API del servicio y las operaciones del modelo, respectivamente.

Respecto a la API, el servicio publicará un subconjunto de la API original, en concreto todas aquellas operaciones que tienen que ver con autenticación, gestión de usuarios y de seguidores.

Método	**URL**	**Comentarios**
`POST`	`/twitter/sessions`	Abre una nueva sesión. Datos (JSON): {email, password} Resultado (JSON): {token, user}
Recurso users		
`GET`	`/twitter/users`	Recupera todos los usuarios del sistema Parámetros: token, opts (JSON) {query, ini, count, sort} Resultado (JSON): [user]

`GET`	`/twitter/users/XXX`	Recupera el usuario con id XXX Parámetros: token Resultado (JSON): user
`POST`	`/twitter/users`	Crea un nuevo usuario Datos (JSON): user Resultado (JSON): user
`PUT`	`/twitter/users/XXX`	Actualiza el usuario con id XXX Parámetros: token Datos (JSON): user Resultado (JSON): user
`DELETE`	`/twitter/users/XXX`	Elimina el usuario con id XXX Parámetros: token
Recurso following/followers		
`GET`	`/twitter/users/XXX/following`	Recupera todos los usuarios a quien sigue el usuario con id XXX Parámetros: token, opts (JSON) {query, ini, count, sort} Resultado (JSON): [user]
`POST`	`/twitter/users/XXX/following`	Añade un nuevo usuario a la lista de usuarios seguidos Parámetros: token Datos (JSON): {id}
`DELETE`	`/twitter/users/XXX/following/YYY`	Elimina al usuario YYY de la lista de usuarios seguidos del usuario XXX Parámetros: token
`GET`	`/twitter/users/XXX/followers`	Recupera todos los usuarios que siguen al usuario XXX Parámetros: token, opts (JSON) {query, ini, count, sort} Resultado (JSON): [user]

Para implementar la API podemos reaprovechar la mayor parde de las operaciones que ya se implementaron en los ficheros *server.js* y *model_mongo.js*. Copiamos todo el código que tiene relación con la creación del servidor Express y con la autenticación, gestión de usuarios y de seguidores en los ficheros *users.js* y *model_users.js*.

Como ahora disponemos de dos microservicios independientes, si queremos arrancarlos en la misma máquina será necesario arrancarlos en puertos diferentes. Además, cada servicio se puede conectar potencialmente a un servidor MongoDB diferente. Para ello, permitiremos especificar tanto el puerto por el que escucha el servicio como la URL del servidor MongoDB, por medio de argumentos en línea de comandos. Podremos arrancar el microservicio por consola así:

```
> node users <port> <url_mongo>
```

Permitiremos también valores por defecto. En el módulo *users.js* añadiremos el siguiente código:

```
let port = 8080;
if (process.argv.length > 2) port = Integer.parseInt(process.argv[2]);
...
app.listen(port);
console.log('Service users listening on port ' + port);
```

Haremos algo parecido en el módulo *model_users.js*.

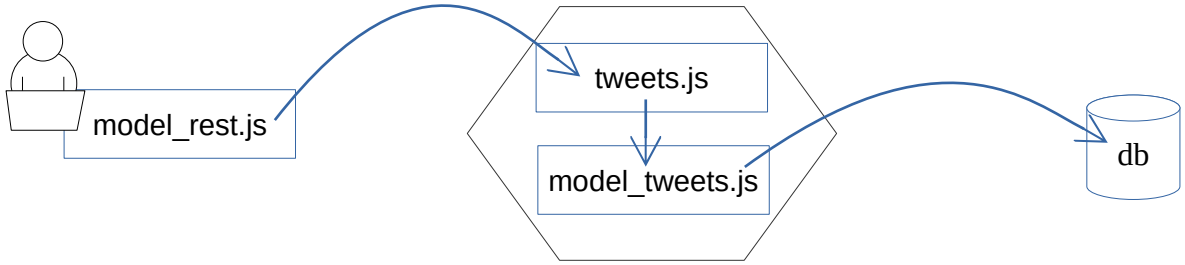
```
let url = 'mongodb://localhost:27017';
if (process.argv.length > 3) url = process.argv[3];

console.log('Using MongoDB url ' + url);
```

Una vez hemos finalizado podemos arrancar nuestro microservicio y comprobar que las operaciones funcionan correctamente.

2.2 Microservicio tweets

En esta sección crearemos el microservicio *tweets*:



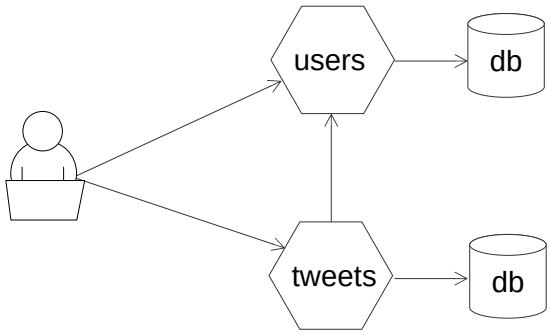
El primer paso consiste en crear los nuevos módulos *tweets.js* y *model_tweets.js*. En este caso, la API a implementar incluye las operaciones sobre tweets, retweets, likes y dislikes:

Recurso tweets		
`GET`	`/twitter/tweets`	Recupera todos los tweets del sistema Parámetros: token, opts (JSON) {query, ini, count, sort} Resultado (JSON): [tweets]
`GET`	`/twitter/tweets/XXX`	Recupera el tweet con id XXX Parámetros: token Resultado (JSON): tweet
`POST`	`/twitter/tweets`	Crea un nuevo tweet Datos (JSON): tweet Resultado (JSON): tweet
`POST`	`/twitter/tweets/XXX/retweets`	Crea un nuevo retweet a partir del tweet con id XXX Parámetros: token Resultado (JSON): tweet
Recurso likes/dislikes		
`POST`	`/twitter/tweets/XXX/likes`	Añade un like al tweet con id XXX Parámetros: token
`POST`	`/twitter/tweets/XXX/dislikes`	Añade un dislike al tweet con id XXX Parámetros: token

A pesar de que podremos reaprovechar gran parte del código de *server.js* y *model_mongo.js,* la implementación de este microservicio será más complicada, debido a dos causas:

1. Todas las operaciones que trabajan con tweets deben ser previamente autorizadas, pero ahora la información sobre autorización reside en el servicio *users*.
2. Las operaciones que devuelven tweets incluyen información sobre los autores de los tweets, pero ahora esta información reside en el servicio *users*.

En ambos casos, el servicio *tweets* necesitará contactar con el servicio *users*. Por lo tanto, el nuevo esquema de microservicios quedaría como sigue:



En las siguientes secciones sugerimos cómo resolver estos dos problemas.

2.2.1 Autorización de operaciones

Para resolver este problema debemos comunicarnos con el microservicio *users*, y consultar si el token es correcto, y en caso de serlo, solicitar la información del usuario que abrió la sesión. Para ello, debemos incluir una nueva operación en la API del microservicio *users.* Proponemos algo como esto:

Método	**URL**	**Comentarios**
`GET`	`/twitter/sessions`	Recupera una sesión previamente creada. Parámetros: token Resultado (JSON): user

En *users.js* incluiríamos un código similar a lo siguiente:

```
app.get('/twitter/sessions', function (req, res) {
    console.log('checkToken ' + req.query.token);
    let opts = { id: req.query.token };
    model.listUsers(req.query.token, opts, (err, users) => {
        if (err) {
            console.log(err.stack);
            res.status(400).send(err);
        } else if (!users.length) {
            res.status(401).send();
        } else {
            res.send(users[0]);
        }
    });
});
```

En *model_tweets.js* debemos incluir todas las operaciones que trabajan con tweets, retweets, likes y dislikes: *listTweets(), addTweet(), addRetweet(), like(), dislike()*. En estas operaciones, lo primero que se hace es verificar que el usuario dispone de autorización, comprobando el token. Proponemos crear una nueva función interna, getSession(), con un contenido similar al siguiente:

```
const axios = require('axios');
function getSession(token, cb) {
  axios.get(urlUsers + '/sessions',
      {
          params: { token: token }
      })
```

```
      .then(res => {
          cb(null, res.data)
      })
      .catch(err => {
          cb(err);
      });
}
```

Esta función contactará con el microservicio users, ubicado en la URL *urlUsers* y verificará que el token es correcto. A cambio, recibirá la información del usuario que abrió la sesión.

2.2.2 Incluir información sobre autores

Cuando se listan los tweets, es necesario obtener información acerca de los autores. Esta información ahora reside en el microservicio *users*, y se publica por medio de la operación HTTP GET sobre la URL `/twitter/users`. Aprovecharemos por tanto esta operación para recuperar dicha información. Para ello, proponemos implementar una nueva operación interna en *model_tweets.js*:

```
function listUsers(token, opts, cb) {
  axios.get(urlUsers + '/users',
      {
          params: { token: token, opts: JSON.stringify(opts) }
      })
      .then(res => {
          console.log(res.data);
          cb(null, res.data)
      })
      .catch(err => {
          cb(err);
      });
}
```

Esta operación será utilizada desde el interior de la operación *listTweets()* para recuperar la información de los autores.

2.2.3 Arrancar el microservicio

Por último, recordar que ahora disponemos de varios servicios que deben arrancarse en puertos diferentes, y contactar con servidores MongoDB distintos. Además, en el caso del microservicio *tweets*, debe contactar con el microservicio *users*, de manera que debe conocer su ubicación. Configuraremos todos estos aspectos por medio de línea de comandos. De este modo, podremos arrancar el microservicio por consola así:

```
> node tweets <port> <url_users> <url_mongo>
```

Permitiremos también valores por defecto. En *tweets.js* incluiremos algo parecido a lo siguiente:

```
var port = 8080;
if (process.argv.length > 2) port = Number.parseInt(process.argv[2]);
...
app.listen(port);
console.log('Service tweets listening on port ' + port);
```

En el módulo *model_tweets.js* incluiremos algo similar a lo siguiente:

```
let urlUsers = 'http://localhost:8080/twitter';
if (process.argv.length > 3) urlUsers = process.argv[3];

let url = 'mongodb://localhost:27017';
```

```
if (process.argv.length > 4) url = process.argv[4];

console.log('Using users url ' + urlUsers);
console.log('Using MongoDB url ' + url);
```

A continuación arrancaremos los dos microservicios y comprobaremos que funcionan correctamente.

2.3 El CLI

Deberemos actualizar convenientemente las URLs de la aplicación cliente cli.js, para que ahora, en lugar de acceder a un único servicio, acceda a dos microservicios ubicados en dos lugares diferentes.

3. Contenerizar microservicios

En este apartado vamos a contenerizar todos nuestros microservicios. Primero, arrancaremos un registro de imágenes Docker privado, donde registraremos las imágenes de nuestros microservicios. Después definiremos una estructura de proyecto adecuada y dockerizaremos los microservicios *users* y *tweets* utilizando ficheros *Dockerfile*. Después desplegaremos toda nuestra aplicación por medio de Kubernetes.

3.1 Registro privado

Antes que nada arrancaremos un registro privado de imágenes Docker. En este registro depositaremos nuestras imágenes. De este modo, posteriormente podremos crear contenedores partiendo de estas imágenes en Kubernetes.

Para arrancar el registro privado basta con ejecutar el siguiente comando:

```
> docker run -d -p 5000:5000 --name registry registry:2
```

3.2 Estructura del proyecto

Vamos a crear una estructura de proyecto que contenga todas las piezas de nuestra aplicación. Para ello, partiendo del proyecto raíz, estructuraremos nuestros fuentes del siguiente modo:

```
/
  |- client
    |- cli.js
    |- package.json
  |- users
    |- users.js
    |- model_users.js
    |- package.json
    |- Dockerfile
  |- tweets
    |- tweets.js
    |- model_tweets.js
    |- package.json
    |- Dockerfile
  |- app.yml
```

Como se puede observar, el CLI residirá ahora en el directorio *client*. Cada microservicio estará incluído en un directorio diferente, donde ubicaremos sus fuentes y el fichero *Dockerfile*, que nos

permitirá generar su imagen. En el directorio raíz residirá el fichero *app.yml* que nos permitirá desplegar toda la aplicación en Kubernetes.

3.3 Creación de contenedores

Comenzaremos con el servicio *users*. En el directorio */users* incluiremos los módulos *users.js* y *model_users.js*. Además, abriremos una consola en dicho directorio y generaremos el fichero *package.json*:

```
> npm init
```

A continuación instalaremos las dependencias necesarias para el microservicio:

```
> npm install express
> npm install mongodb
```

El último paso consistiría en generar el fichero *Dockerfile,* que determinará cómo se creará la imagen. Partiremos de la imagen *node* disponible en el Docker Hub, y que incluye una instalación de Node.js. Incluiremos algunas etiquetas descriptivas. Nuestro contenedor aceptará dos parámetros de configuración, el puerto por el que escucha y la URL del servidor MongoDB al que se conectará. Esto lo definiremos por medio de variables de entorno. Finalmente, incluiremos todos los recursos necesarios para que el microservicio funcione en el directorio raíz */app* del contenedor, instalaremos las dependencias en el interior del contenedor y definiremos su punto de entrada. Sería algo parecido a esto:

```
FROM node
LABEL description="Users microservice"
LABEL version="1.0"
LABEL maintainer="tumail@upv.es"

ENV PORT=8080
ENV URL_MONGODB="mongodb://127.0.0.1:27017"

ADD users.js model_users.js package.json /app/
RUN npm --prefix /app install
ENTRYPOINT node /app $PORT $URL_MONGODB
```

El siguiente paso consistiría en generar la imagen del microservicio, y registrarla en el registro privado. Para ello nos ubicamos en el directorio */users* y ejecutamos los siguientes comandos:

```
> docker build . -t localhost:5000/users
> docker push localhost:5000/users
```

Esto generará una imagen con la etiqueta *localhost:5000/users* y la registrará en el registro de imágenes privado. A partir de este momento ya podemos ejecutar nuestro nuevo contenedor. Con esta configuración, hay que tener en cuenta que por deecto el microservicio publicará su API en el puerto 8080 y se conectará al servidor MongoDB disponible en la URL *mongodb://127.0.0.1:27017*. Para comprobar que todo funciona correctamente podemos hacer un “truco” temporal, y compartir la red del host con la del contenedor (opción *network=host*):

```
> docker run --name=users --rm -d --network=host localhost:5000/users
```

Con este truco el contenedor estará visible en el puerto 8080 y además se conectará a un servidor MongoDB que está corriendo en *127.0.0.1:27017*.

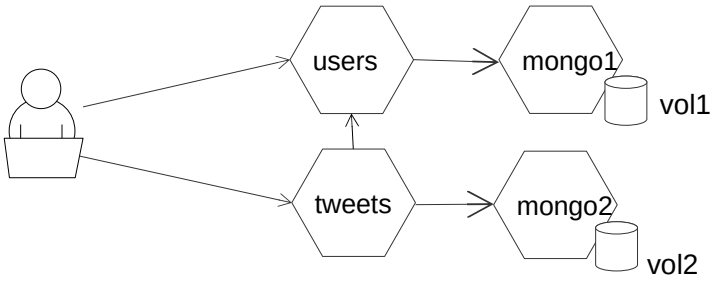
Podemos comprobar que todo funciona correctamente, y podemos observar los logs de nuestro microservicio con:

```
> docker logs users
```

El siguiente paso consistiría en hacer algo parecido con el microservicio *tweets*. **Se deja este ejercicio al alumno.**

4. Despliegue en Kubernetes

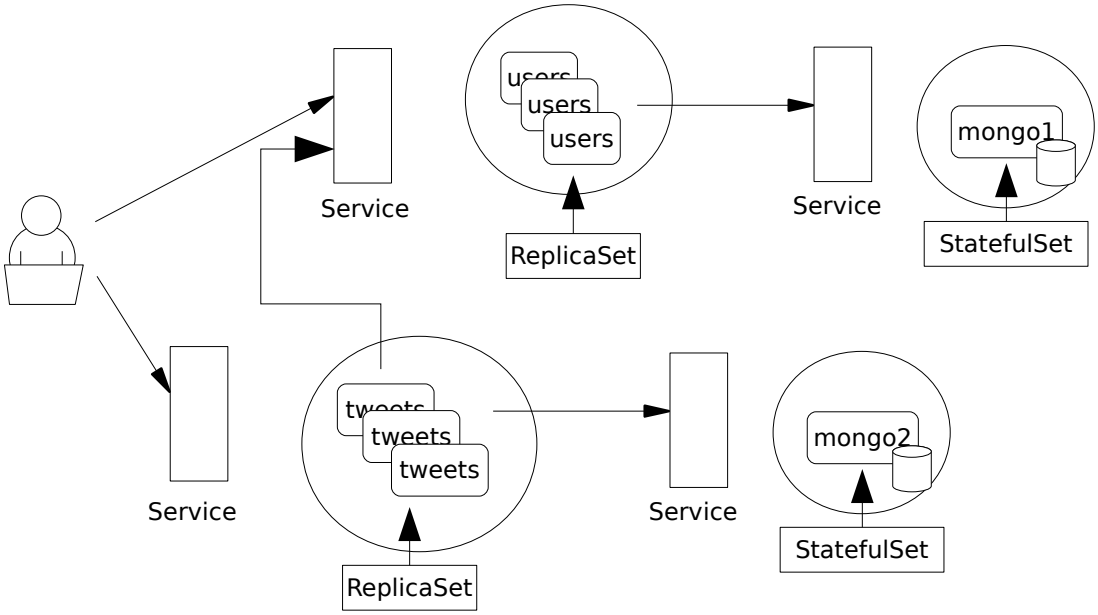
El último paso en el laboratorio consiste en desplegar todos los microservicios de nuestra aplicación en Kubernetes. En principio, se pretende construir una arquitectura como la que se muestra en la siguiente figura:



Como ya sabemos, en Kubernetes cada microservicio será encarnado por un pod. Los microservicios *users* y *tweets* son sateteless, es decir, no mantienen estado y por lo tanto pueden ser replicados tantas veces deseemos. Sin embargo, los microservicios *mongo1* y *mongo2* sí poseen estado, son microservicios stateful, y replicarlos implica configuraciones más complejas de MongoDB.

Para simplificar, proponemos gestionar los microservicios stateless por medio de controladores ReplicaSet y los microservicios stateful por medio de controladores StatefulSet. Además, como algunos microservicios dependen de otros deberemos habilitar descubrimiento dinámico por medio de *Services*.

A continuación se presenta un esquema de esta posible configuración.



Se deja este ejercicio al alumno.