Desarrollo de técnicas de
aprendizaje entre pares.

“noestudiosolo”

Documentación Backed

Plataforma enfocada al desarrollo de técnicas de aprendizaje entre pares, una puesta en marcha para la digitalización de la sala de clase tradicional.

Waldo Ancacoy - Bruno Binner - Michel Muñoz.
Taller de Integración IV. Prof. Julio Rojas.
Coordinadores: Colegio de Ayudantes UCT.

1. Servidor

En esta primera parte se documenta todo lo relacionado con el sistema de soporte que sostiene el software de la API así como el sistema de base de datos que hace funcionar al backend.

Primero hay que describir el hardware físico y sus propiedades. Actualmente el servidor es una máquina virtual que se encuentra en las dependencias de la escuela. El servidor virtual que se usa es el llamado 'Servidor de proyectos' que la escuela provee para que los alumnos desarrollen sus sistemas con un hardware predispuesto y se ahorran la necesidad de contratar servicios externos como VPS. Estos servicios son prestados y administrador desde el Laboratorio de Administración de Sistemas (LASS) que es una organización dependiente de la escuela y que dirige el profesor Alejandro Mellado.

Dado a que uno de los miembros del backend es también al mismo tiempo del LASS se aprovechó de usar. Este se encuentra en la máquina anfitrión de la dirección 192.168.4.5 interna de la escuela y dentro de este está la máquina Nepen con dirección pública en 164.77.114.250.

Características del S.O:

- Sistema Operativo FreeBSD 11.2 stable release.
- Servicio HTTP con Nginx.
- Servidor web con virtual host en dominio noestudiosolo.inf.uct.cl
- Servicio SSH, MongoDB, etc.

El diagrama de integración de sistemas ayudará a entender de mejor forma cómo es que funciona el sistema subyacente del proyecto.

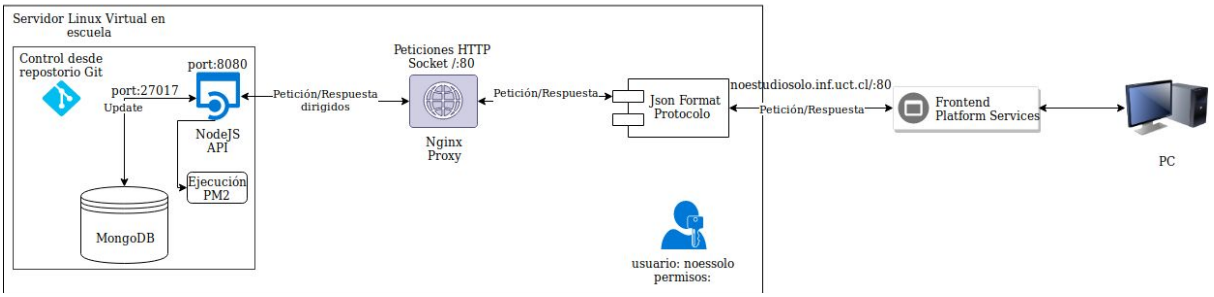


Diagrama de Integración de Sistemas Final

1.1 Configuración hecha y dependiente del LASS

Dado a que el sistema es prestado y se comparte con más estudiantes es que es administrado en sus permisos por permitir una buena configuración entre diversos proyectos sin que estos se vean alterados. Es por esto que las siguientes configuración son dependientes del LASS y fueron pedidas a media que fueron necesitadas y probado su funcionamiento para el proyecto fueron utilizadas:

1.2 Respecto al usuario del proyecto noestudiosolo

Lo primero fue crear el usuario su nombre es “noessolo” y su clave es “notalone2018”. Este fue creado con permisos generales (no administrador) del sistema por lo cual no puede ejecutar operaciones que cambien el funcionamiento general del sistema operativo.

1.3 Configuración del servicio web para el usuario noessolo

Este es configurado como host virtual en el servidor mediante Nginx y un subdominio registrado por en la escuela de informática. Este incluye que lo que esté dentro de la carpeta /home/noessolo/www será mostrado a través de internet si es buscado por la dirección noestudiosolo.inf.uct.cl que fue registro por la wildcard (subdominios) que poseen en *.inf.uct.cl donde esta el proyecto. Está configuración incluye también un proxy de Nginx que permite redirigir las conexiones realizadas en el puerto 8080 que es donde el socket de la API trabaja y luego pasa por el puerto 80 (defecto HTTP) vía Nginx. Permitiéndole tener una capa de seguridad incluyendo HTTPS y restringiendo los permisos de ejecución del usuario antes mencionado, aumentando considerablemente la seguridad.

La configuración está hecha en /usr/local/etc/nginx/sites-available/noestudiosolo.inf.uct.cl y es la siguiente:

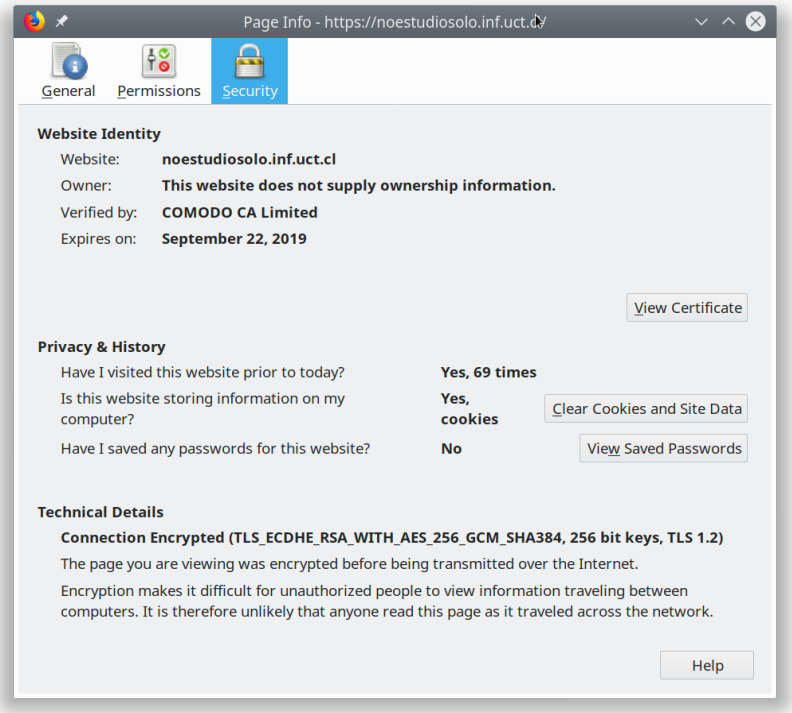
```
server {
  listen     164.77.114.250:80;          #IP pública de la escuela
  server_name  noestudiosolo.inf.uct.cl;    #El servidor virtual respondera al
dominio
  rewrite    ^(.*) https://noestudiosolo.inf.uct.cl$1 permanent;
}

server {
       listen       164.77.114.250:443;
       server_name  noestudiosolo.inf.uct.cl;
```

```
        root   /usr/home/noestudiosolo/www;
        access_log  /var/log/nginx/noestudiosolo.inf.uct.cl.access.log;
        error_log   /var/log/nginx/noestudiosolo.inf.uct.cl.error.log;
             #Configuración HTTPS
        ssl on;
        ssl_certificate     /usr/local/etc/ssl/webcert/inf.uct.cl.crt;
        ssl_certificate_key /usr/local/etc/ssl/webcert/inf.uct.cl-nopass.key;

        ssl_session_timeout  5m;
        ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;

        location / {
             #Configuración de Proxy nginx
              proxy_pass          http://localhost:8080/;
              proxy_set_header    Host $host;
              proxy_pass_header Server;
              # be carefull, this line doesn't override any proxy_buffering on
set in a conf.d/file.conf
              proxy_buffering off;
              proxy_set_header X-Real-IP $remote_addr;  #
http://wiki.nginx.org/HttpProxyModule
              proxy_set_header Host $host;  # pass the host header
              proxy_http_version 1.1;  # recommended with keepalive connections
              # WebSocket proxying - from
http://nginx.org/en/docs/http/websocket.html
              #proxy_set_header Upgrade $http_upgrade;
              #proxy_set_header Connection $connection_upgrade;
        }
}
```



1.4 Configuración de los paquetes necesarios

Se solicitó además los paquetes necesario al LASS como lo son el entorno NodeJS y la base de datos MongoDB, este fue instalada y fueron dados los permisos necesarios para ser ejecutada dentro de la carpeta del usuario noessolo. Estas instalaciones son triviales y fueron hechas a partir de los comandos `pkg install node npm mongodb mongodb-utils` esto previo a su búsqueda con el parámetro `pkg search`.

Una vez listos hubo hacer una configuración de mongodb que por defecto no permite conexiones que no sean de la dirección local negando el acceso externo. Es por ello que se cambió su archivo de configuración en `/usr/local/etc/mongodb.conf` y se tuvo que dejar el parametro de bindIP en 0.0.0.0 (cualquier conexión IP) y authorization 'enabled'. Ya que al permitir estas conexiones debemos agregar la seguridad de autenticación en el servicio MongoDB. Usualmente también se cambia el puerto por defecto por otro, pero no fue considerado dado que su utilidad no es tan tanta.

```
# network interfaces
net:
  port: 27017
  bindIp: 0.0.0.0
 #bindIp: 127.0.0.1  # Listen to local interface only, comment to listen on all interfaces.

security:
```

  authorization: 'enabled'
#operationProfiling:

1.5 Configuración de usuarios en MongoDB

Una vez establecido la seguridad de autenticación para los usuarios de mongoDB es que se debió crear un usuario para el proyecto correspondiente y luego para los grupos de front end probaran sus sistemas iniciales, también finalmente se creó una base de datos de prueba para ellos y su usuario correspondiente.

Con los permisos de administrador de MongoDB que posee el LASS se ejecutó las siguientes instrucciones en mongod (la shell de MongoDB).

```
> use noestudiosolodb
> db.createUser(
{
    user: "noessolo",
    pwd: "notalone",
    roles: ["readWrite"]
})
```

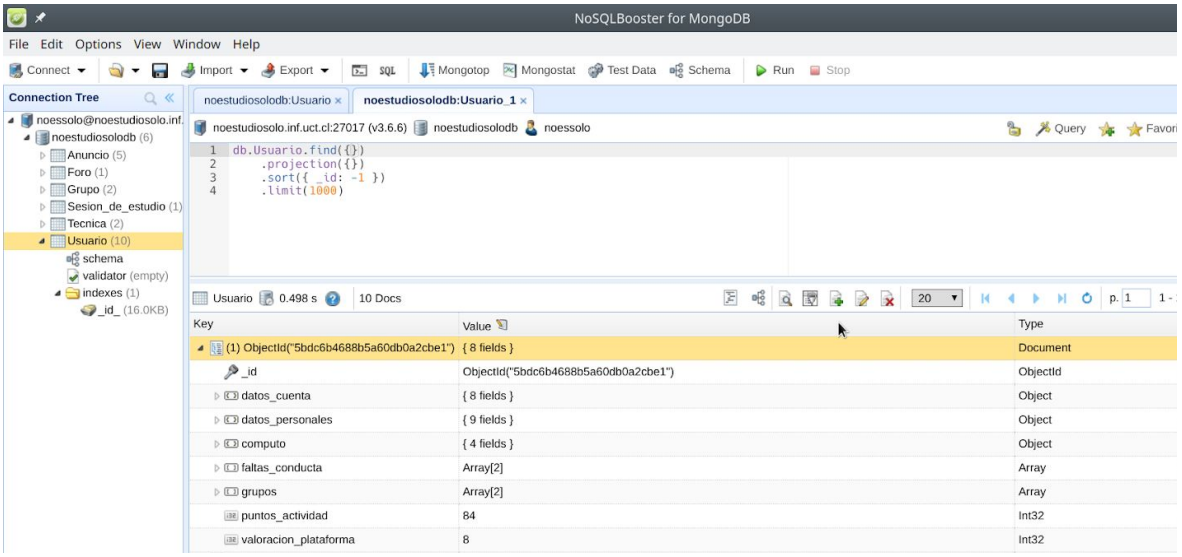
Al ejecutar *use* se crea la base de datos con el nombre. Luego se le agrega el usuario con sus permisos de lectura y escritura solo esa base de datos. Una vez creado el usuario se puede autenticar en el sistema con la IP del servicio y los parámetros necesarios. Con el siguiente comando se puede conectar con el cliente de mongo:

```
mongo -u noessolo -p notalone2018 --authenticationDatabase noestudiosolodb
```

Una vez dentro hay que ejecutar *use noestudisolodb* y tendremos todos los permisos de uso, creación de listas y colecciones, etc. Un proceso similar utilizan en la API para conectarse a la base de datos.

Destacar con el paquete `mongodb-utils` tenemos permiso de uso de `mongodump` y `mongorestore` que se utilizan para cargar o exportar la base de datos. Estos comandos se utilizan como usuario del sistema y no desde la shell mongod.

Software con utilidades gráficas también pueden ser usados. Por ejemplo aquí se usó el NoSQLBooster for MongoDB.



1.5 Configuración y uso del usuario del proyecto noessolo

1.5.1 Comandos para uso del servidor Freebsd

En este caso fue montado el servidor en un sistema Freebsd, en el cual para poder movernos dentro de él, es necesario tener un manejo básico de comandos estándar de linux (más conocido), con algunas pequeñas variaciones pero que en su mayoría no afectan en lo que se realiza dentro del servidor.

En el caso de querer instalar algunos paquetes necesarios, se puede realizar con el siguiente comando el cual difiere de lo que es linux.

La conexión al servidor se realizó por medio de SSH a nuestro dominio:

```
noessolo@noestudiosolo.inf.uct.cl
```

Para poder trabajarlo fue necesario los siguientes comandos basicos.

Para listar el contenido de un directorio y ficheros.

```
ls /home/directorio
```

se puede usar *-a* para mostrar los archivos ocultos y *-l* para mostrar los usuarios, permisos y la fecha de los archivos.

cd (cambiar directorio), es como su nombre lo indica el comando que necesitarás para acceder a una ruta distinta de la que se encuentra.

```
cd /home/carpeta
```

Para subir un nivel en la carpeta que se ubique

```
cd ..
```

Rm (remover), es el comando necesario para borrar un archivo o directorio.

```
rm -r /home/ejem-mkdir /home/carpeta
```

Mkdir (crear directorio), crea un directorio nuevo tomando en cuenta la ubicación actual.

```
mkdir /home/carpeta
```

Mv (mover), mueve un archivo a una ruta específica, y a diferencia de *cp*, lo elimina del origen finalizada la operación.

```
mv /home/ejemplo.txt  /home/carpeta/ejemplo2.txt
```

1.5.2 Manejo y conexión con github.

Inicialmente es necesario realizar carga de los archivos a utilizar en el servidor, esto incluye las carpetas a necesitar junto con los archivos. Para ello clonamos el directorio así dentro de él ir actualizando constantemente los cambios realizados desde git.

Clonar un directorio git en el servidor:

```
git clone https://github.com/wanca02/noestudiosolo
```

Es posible también realizar commit desde el mismo servidor si así se deseara.

Realizar un commit desde el servidor:

```
git add <nombre del archivo>
```

```
git commit -m "comentario del commit"
```

Es posible cargar de nuestro servidor a github alguna modificación por medio del siguiente comando, enviando directamente a alguna rama específica.

Push al directorio:

```
git push origin <nombre rama>
```

Dentro del repositorio creado en el servidor, por defecto lo mostrado y cargado siempre se encontrará en la rama master, a menos que se indique lo contrario, por esto es necesario ser cuidadosos en su manejo y utilizar la rama donde tenemos los datosa usarse.

Cambio de rama del directorio:

```
git checkout -b <nombre rama>
```

De igual forma que al usar push es necesario actualizar en nuestro servidor los cambios que se van realizando por ello también necesitamos hacer la carga de archivos desde github hasta nuestro repositorio, ahora podemos hacer un pull general.

Actualizar directorio:

```
git pull
```

Además del pull general podemos realizar uno de manera más específica indicando cual rama se desea dejar con lo último subido al git.

Actualizar rama:

```
git pull origin <rama especificada>
```

Importar base de datos desde el directorio git:

```
mongorestore --dir <carpeta clonada de git> -d <base de datos> -u
<usuario> -p <password>
```

Comando utilizado en nuestro caso:

```
mongorestore --dir noestudiosolo -d noestudiosolodb -u noessolo -p
notalone2018
```

Exportar base de datos, para respaldo:

```
mongodump -d <nombre base de datos> -u <usuario> -p <password> -o
<directorio donde guardar la exportación>
```

Para poder manejar un servidor git es necesario poder exportar un repositorio existente a un nuevo repositorio básico, a un repositorio sin carpeta de trabajo por convenio, los nombres de los repositorios básicos suelen terminar en '.git'. Esto básicamente para tener un mejor orden y acceso para los participantes, donde puedan colocar las actualizaciones necesarias

```
-git clone --bare noestudiosolo noestudiosolo.git
```

En caso de ser necesaria ayuda para realizar alguna sentencia es posible usar

```
-git --help
```

Este Imprime la sinopsis y una lista de los comandos más utilizados. Si se da la opción --all o -a se imprimen todos los comandos disponibles. Si se nombra un comando Git, esta opción abrirá la página de manual para ese comando.

1.5.3 Uso de PM2 y Mongoose

PM2 es un paquete descargado desde NPM (gestor de paquetes de Node) y nos permite ejecutar la API es segundo plano como servicio. Su uso es bastante sencillo en vez de ejecutar `node app.js` se realiza un pm2 `start app.js.` Otros parámetros son *stop, reload, show, etc.* También sirve para hacer entornos de producción, desarrollo y configurar a nuestro gusto el funcionamiento de la API como servicio.

Mongoose también fui instalado vía NPM. Luego fue conectado mediante el código necesario que está descrito en documentación de API. Con esto, teníamos listo el driver que conecta Node con MongoDB.

1.5.4 Control de repositorio por git

Como ha sido explicado podemos inferir que mediante git estamos actualizando nuestras versiones de la API como la de base de datos que luego es cargada. Osea el proceso de cambio de versión incluye los siguientes pasos:

- Parar la API en ejecución PM2 stop app.js
- Actualizar con git los cambios desde repositorio
- Respaldar la base de datos y cargar la nueva
- Volver a ejecutar y probar con PM2 start app.js

Es así como se lleva el flujo desde el usuario. Este se preocupa de ejecutar el software que tiene permiso de usar y con eso es suficiente para satisfacer las necesidades de control del sistema y su ejecución. Mucho de lo realizado se puede buscar en internet y está tal cual descrito y más a profundidad. También otras necesidades pueden ser pedidas al LASS o al profesor Mellado y serán atendidas. Con esto concluye la documentación de servidor.

2. Base de datos

Para el proyecto se utilizó un motor de base de datos no relacionales MongoDB Community Edition versión 3.6.6 en el servidor y 4.0.4 en el cliente (aunque en el cliente es probable que funcione con la última versión disponible de mongo.
Se utilizó esta clase de base de datos debido a que los datos que deben ser almacenados por la plataforma pueden variar dependiendo de la técnica de estudio que utilice la sesión.
Documentación sobre como instalar MongoDB apropiadamente se encuentra aquí:
Windows
Linux
Durante el transcurso del proyecto no fue necesario utilizar un gestor con interfaz visual de mongo pero se puede optar a utilizar MongoDB Compass.

2.1 Uso de mongo como cliente

El uso de mongo se hace a través de la consola o terminal con el comando `mongo`, tanto en windows como linux. Al hacer eso se ingresa al entorno de mongo donde se pueden administrar el motor. También existen otros comandos que se realizan desde fuera de mongo, o sea desde la terminal o consola, siendo los principales los para importar y exportar bases de datos.
Para conectarse a mongo de forma remota al servidor se utiliza mongo con los siguientes parámetros.

```
mongo --host noestudiosolo.inf.uct.cl --port 27017 -u
noestudiosolo_front -p noestudiosoloforfrontend --authenticationDatabase
```

```
noestudiosolodb_front
```

2.2 Restauración de la base de datos

La restauración de la base de datos utilizada por el proyecto que se encuentra en el repositorio del proyecto, debe hacerse a través de la terminal o consola con el comando `mongorestore`. A continuación su uso con los parámetros correspondientes al caso presente. Teniendo `dump/` como la ruta a la carpeta conteniendo la base de datos.

```
mongorestore --db noestudiosolodb_front dump/
```

2.3 Estructura de la base de datos

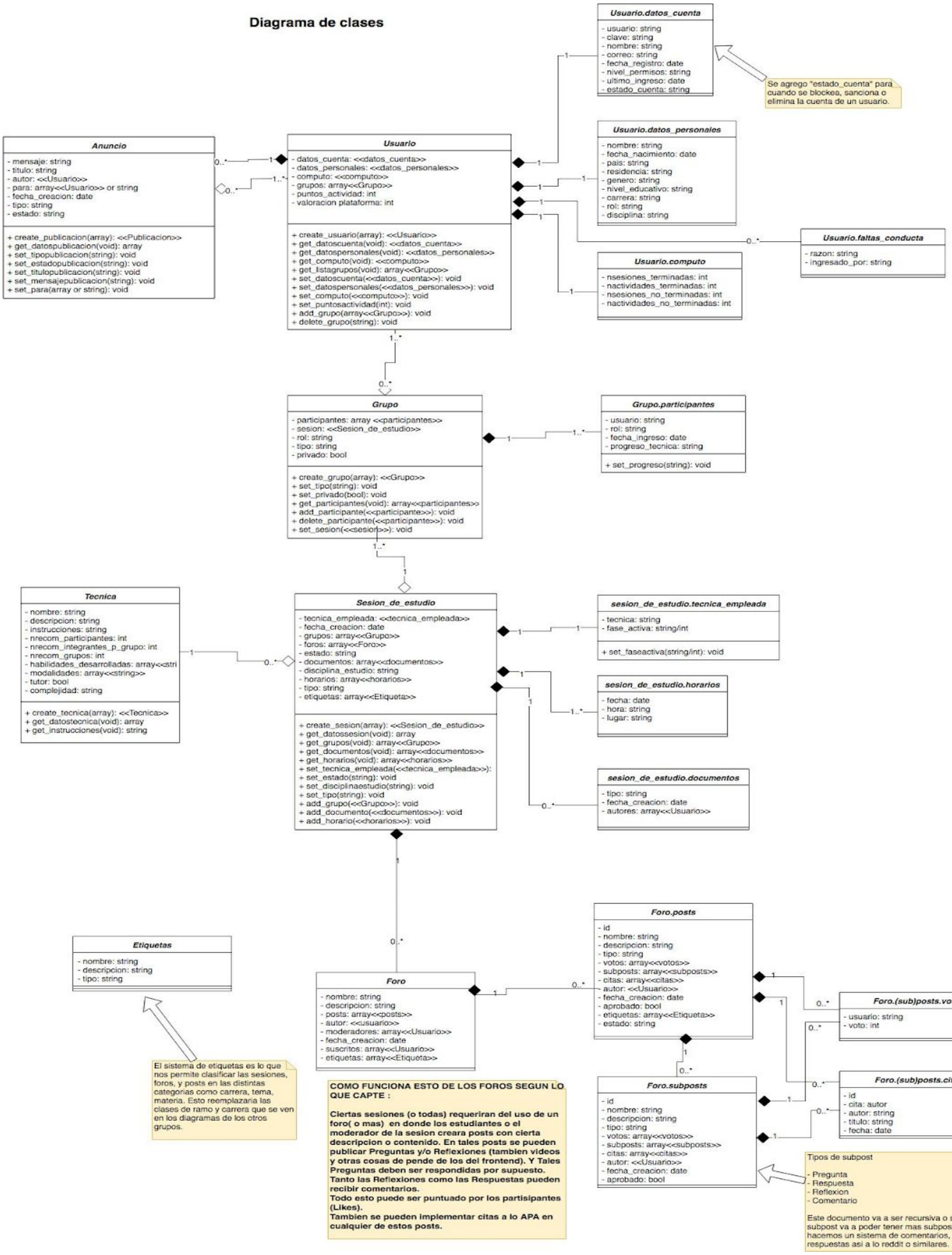
La estructura de la base de datos se puede encontrar en el diagrama de clases donde se definieron las colecciones: ***Usuario***, ***Anuncio***, ***Grupo***, ***Tecnica***, ***Sesion_de_estudio***, ***Foro***, *y* ***Etiquetas***, cada una de esas colecciones posee su propia clase. Las otras clases presentes en el diagrama son clases que están presentes dentro de las mencionadas anteriormente, estas clases poseen en el nombre de estas la notación ***<clase_padre>.<clase>*** esto se hizo para representar documentos que estuvieran guardados de forma anidada, o sea documentos dentro de otros documentos. Un ejemplo de esto sería ***Usuario.datos_cuenta*** donde ***Usuario*** es una de las colecciones de la base de datos, y ***datos_cuenta*** es un documento dentro de esta colección que posee más documentos dentro de sí como el nombre de usuario, su clave, y otros datos relevantes para la cuenta del usuario.

En el diagrama no se especifica la campo **_id** ya que mongo lo crea de forma automática al crear un documento, en los casos que este se especifique quiere decir que tal clase posee una _id pero no es generada de forma automática por mongo por lo que hay que tener cuidado de crearles una _id. Este caso de da en los documentos anidados.
También aclarar que cuando un documento es de tipo colección, por decir asi Usuario, quiere decir que ese documento se utiliza para hacer una referencia a tal colección desde la _id. Por ejemplo si tenemos el documento ***grupos: array<<Grupo>>*** en la colección ***Usuario***, quiere decir que el documento ***grupos*** guardará un arreglo con una lista de IDs que referencian a la colección ***Grupo***.

No confundir con documentos como ***datos_cuenta: <<datos_cuenta>>*** en ***Usuario***, esto solo hace referencia a que ese documento es de tipo ***datos_cuenta*** que es un documento contenido dentro de este mismo.
Pueden diferenciar entre estos dos últimos tipos viendo cuales empiezan con mayúsculas después del **<<** y cuáles no.
Después de la presentación del siguiente diagrama se da una explicación detallada de que contiene cada documento de la base de datos. El diagrama lo pueden encontrar en mejor calidad en el repositorio junto con sus archivos html para que lo puedan editar con draw.io.

2.4 Diagrama de clases

Diagrama de clases



2.5 Explicación colecciones y documentos en la base de datos

La estructura con la que se presenta el documento toma en cuenta la colección existente en la base de datos, los documentos y documentos anidados de la forma:

Coleccion

documento_anidado

- **documento(tipo de dato): Descripción del documento**

documento: Descripción del documento.

A continuación la estructura real de la base de datos:

Usuario

datos_cuenta

- **usuario(string):** Nombre de usuario con el que se ingresa a la plataforma.
- **clave(string):** Contraseña privada del usuario para ingresar a la plataforma.
- **nombre(string):** Nombre del usuario a mostrar en la plataforma, así como un alias o apodo.
- **correo(string):** Correo electrónico del usuario.
- **fecha_registro(date):** Fecha en la que creó su cuenta en la plataforma.
- **nivel_permisos(string):** Nivel de permisos del usuario (en caso de tener cuentas de administrador u otros).
- **ultimo_ingreso(date):** Fecha en la que ingresó a la plataforma por última vez.
- **estado_cuenta(string):** Dato para identificar el estado de la cuenta (eliminada, bloqueada, sancionada, normal, etc)

datos_personales

- **nombre(string)**: Nombre real del usuario (nombre y apellidos).
- **fecha_nacimiento(date)**: Fecha de nacimiento del usuario.
- **pais(string)**: Pais de donde es el usuario
- **residencia(string)**: Lugar donde vive el usuario, el usuario especifica.
- **genero(string)**: Genero del usuario.
- **nivel_educativo(string)**: [enseñanza media completa, enseñanza basica completa, etc]
- **carrera(string)**: Carrera a la que pertenece el usuario.
- **rol(string)**: Rol o profesión que ejerce actualmente el usuario. Por ahora debería ser una selección entre los siguientes: [estudiante básica, media, superior; profesor básica, media, superior; administrativo educacional; magíster?; otro (especifica el usuario)].
- **disciplina(string)**: Disciplina a la que el usuario pertenece. Ejemplo: [matematicas, ingenieria, biologia, fisica, etc]

computo

- **nsesiones_terminadas(int)**: Número de sesiones de estudio completadas por el usuario.
- **nactividades_terminadas(int)**: Número de actividades de sesión(actividades que se realizan durante las sesiones de estudio) completadas por el usuario.

- **nsesiones_no_terminadas(int)**: Número de sesiones de estudio no completadas por el usuario.
- **nactividades_no_terminadas(int)**: Número de actividades de sesión no completadas por el usuario.

faltas_conducta(array)

- **razon(string)**: Razón por la que se reporto al usuario.
- **ingresado_por(string)**: Id del usuario que reportó al usuario.
- **estado(string)**: Por si las reportes tienen que pasar primero por un administrador o moderador que los aprueba este dato tomaría los valores de aprobado o desaprobado.

grupos(array<<Grupo>>): Arreglo que guarda una lista de grupos a los cuales el usuario pertenece. Un usuario puede pertenecer a varios grupos.
puntos_actividad(int): Cantidad de puntos que ha ganado el usuario a través de actividades en la plataforma.
valoracion_plataforma(int): [1-5] Nota en estrellas que le da el usuario a la plataforma.

Grupo

participantes (array)

- **usuario(string)**: Usuario que pertenece al grupo.
- **rol(string)**: Rol que cumple el usuario en el grupo (adm, tutor, etc).
- **fecha_ingreso(date)**: Fecha en la que el usuario se unió al grupo.
- **progreso_tecnica(string)**: Fase o etapa en la que el usuario se encuentra respecto a la técnica que está implementando la sesión.

sesion(string): String que enlaza el grupo con la sesión de estudio a través de la id de la sesión.
rol(string): Dato para definir el rol del grupo en una sesión, se puede utilizar para definir el moderador, administrador o ayudante a cargo de una sesión de estudio.
tipo(string): Dato reservado por si es que más adelante se dividen los grupos en categorías.
privado(bool): Para establecer si el grupo es de carácter privado o no (para que usuarios desconocidos no se unan a un grupo privado de por ejemplo amigos o compañeros de ramo).

Etiquetas

nombre(string): Nombre de la etiqueta. Por ejemplo “Calculo I”.
descripcion(string): Descripcion de la etiqueta por si se requiere.
tipo(string): Tipo de etiqueta que sirve para categorizarlas. Un ejemplo podría ser tener etiquetas de tipo: ramo, competencia, materia, o carrera.

Sesion_de_estudio

tecnica_empleada

- **tecnica(<<Tecnica>>)**: Técnica que emplea la sesión de estudio. Una sesión podría poseer más de una técnica pero los participantes deben completar una primero para pasar a la siguiente (esto se podría implementar más a nivel de

aplicación, cosa que cuando se termine la técnica activa se les da la opción de migrar a otra). Además un dato aquí señala en qué fase o etapa de la técnica se encuentra la sesión si es necesario.

- **fase_activa(string/int)**: Fase o etapa en la que se encuentra la sesión de estudio con respecto de la técnica.

documentos (array)

Lista de documentos (guías, videos, pruebas, etc) que sean generados o utilizados para las sesiones de estudio.

- **tipo(string)**: Tipo de documento (pdf, docx, excel, etc).
- **fecha_creacion(date)**: Fecha en el que se subió el documento.
- **autores(array<<Usuario>>)**: Lista de autores del documento.

horarios (array)

Lista de objetos Horario donde se guardan las fechas, horas y lugar en el que será la sesión.

- **fecha(date)**: Fecha y hora de la sesión.
- **ugar(string)**: Lugar de la sesión.

fecha_creacion(date): Fecha de creación de la sesión de estudio.

grupos(array<<Grupo>>): Lista de grupos que están o van a participar de la sesión.

estado(string): Estado en el que se encuentra la sesion (iniciada, en procesos, cancelada, etc).

disciplina_estudio(string): Para guardar la materia que se está estudiando durante la sesión si es necesario.

tipo(string): Dato reservado por si más adelante se dividen las sesiones de estudio en categorías.

modalidad(string): Si es presencial, no presencial o ambas.

etiquetas(array<<Etiqueta>>): Lista que contiene las id de las etiquetas relacionadas a la sesión de estudio. Estas etiquetas puede definir cosas como a qué ramo, carrera, o materia pertenece la sesión de estudio, incluso puede definir las competencias que se trabajan en la sesión.

foros(array<<Foro>>): Lista con las ids de los foros pertenecientes a la sesión de estudio.

Foro

posts (array)

- **votos (array)**
  - **usuario(<<Usuario>>)**: Id del usuario.
  - **voto(int)**: Valor del voto, 1 es un voto positivo, -1 negativo. 0 neutral.
- **citas (array)**
  - **id(string):** id de la cita.
  - **autor(<<Usuario>>)**: Guarda el o los nombres de los autores de lo citado.
  - **titulo(string)**: Titulo del libro, documento citado.
  - **fecha(date)**: Fecha de publicación del documento.
- **subposts (array)**

  Los subpost poseen una estructura casi idéntica a la de un post normal, incluso almacena mas subpost dentro de si mismo, haciéndolo un documento recursivo.

- **votos (array):** Almacenan lo mismo que el votos ya visto.
- **citas (array):** Almacena lo mismo que en el citas ya visto.
- **subposts (array<<subpost>>):** Almacena una lista de subposts de esta misma estructura dentro de sí.
- **id(string):** id del subpost. IMPORTANTE, debido a que los post son documentos almacenados al interior de un documento se requiere que se genere una id para estos ya que mongo no se las genera de forma automática.
- **nombre(string)**: Nombre del subpost
- **contenido(string)**: Contenido del subpost o el cuerpo, puede ser html, etc.
- **tipo(string)**: Tipo de subpost puede ser de tipo comentario, reflexion, respuesta o pregunta. Se puede extender a otros tipos también.
- **autor(<<Usuario>>)**: Usuario que creó el subpost, guarda la id del usuario.
- **fecha_creacion(date)**: Fecha en que se creó el subpost.
- **aprobado(bool)**: Por si se requiere una aprobación/revisión/validación antes de publicar un subpost en el foro este dato indica tal estado.

- **id(string)**: Id del post, IMPORTANTE, debido a que los post son documentos almacenados al interior de un documento se requiere que se genere una id para estos ya que mongo no se las genera de forma automática.
- **descripcion(string)**: Descripcion del post.
- **contenido(string)**: Contenido del post o el cuerpo, puede ser html, etc.
- **tipo(string)**: Tipo de post por si es necesario especificarlo.
- **autor(<<Usuario>>)**: Usuario que creó el post, guarda la id del usuario.
- **fecha_creacion(date)**: Fecha en que se creo el post.
- **aprobado(bool)**: Por si se requiere una aprobación/revisión/validación antes de publicar un post en el foro este dato indica tal estado.
- **estado(string)**: Estado en que se encuentra el post, podría servir para cuando se tiene en modo borrador, publicado, o cerrado.
- **etiquetas(array<<Etiqueta>>)**: Lista que contiene las id de las etiquetas relacionadas a la sesión de estudio. Estas etiquetas puede definir cosas como a qué ramo, carrera, o materia pertenece la sesión de estudio, incluso puede definir las competencias que se trabajan en la sesión.

nombre(string): Nombre del foro.
descripcion(string): Descripcion del foro.
autor(<<Usuario>>): Usuario que creó el foro, guarda la id del usuario.
moderadores(array<<Usuario>>): Lista de los usuarios que poseen permisos especiales sobre el foro, almacena las id.
fecha_creacion(date): Fecha de creación del foro.
suscritos(array<<Usuario>>): Lista de usuarios suscritos al foro, se puede utilizar para un sistema de notificaciones, guarda las id de los usuarios.
etiquetas(array<<Etiqueta>>): Lista que contiene las id de las etiquetas relacionadas a la sesión de estudio. Estas etiquetas puede definir cosas como a qué ramo, carrera, o materia pertenece la sesión de estudio, incluso puede definir las competencias que se trabajan en la sesión.

Tecnica

computo

- **veces_utilizada(bool)**: Número de veces sesiones han implementado la técnica por completo terminando todas las actividades.
- **veces_abandonada(bool)**: Número de veces que sesiones han abandonado la técnica, o sea nunca terminaron las actividades.

nombre(string): Nombre de la técnica.
descripcion(string): Descripcion de la técnica.
descripcion_breve(string): Descripcion breve de la técnica.
instrucciones(string): Instrucciones a seguir para implementar la técnica en la sesión de estudio. Aqui podrian ir incluida las actividades.
nrecom_participantes(int): Numero recomendado de participantes para la sesión que implementa la técnica.
nrecom_integrantes_p_grupo(int): Numero recomendado de integrantes por grupo de la sesión.
nrecom_grupos(int): Numero recomendados de grupos a formar para la sesión.
competencias_desarrolladas(array<<string>>): Lista de habilidades o competencias que desarrolla en los participantes la técnica (expresión oral, trabajo en equipo, resolución de problemas, etc).
modalidades(array<<string>>): Modalidades en las que trabaja la tecnica (investigacion, debate, foro, etc).
tutor(bool): Dato booleano que nos dice si es necesario un tutor o no para implementar la técnica.
complejidad(string): Nivel de complejidad para implementar la técnica si es que existe alguno.
presencial(bool): Dato que dice si es necesario que la técnica sea implementada de forma presencial.
disciplinas(array<<string>>): Disciplinas a las que se adapta mejor la técnica.

Anuncio

Esta colección almacena algo así como “mensajes” que circulan dentro de la plataforma. El objetivo inicial de esta colección es para proveer de una forma de hacer llegar “Anuncios” a las páginas de inicio del usuario por parte de los administradores, pero se podría expandir y crear publicaciones de administradores de sesión a los participantes de tal sesión. Es por esta posibilidad que el dato “**para(array<<Usuario>> or string)**” se dejó como un arreglo de usuarios, o un string que podría representar un grupo definido de usuarios como “TODOS”.
mensaje(string): Cuerpo de la publicación.
titulo(string): Titulo de la publicación.
autor(<<Usuario>>): Autor de la publicación (id del usuario).
para(array<<Usuario>> or string): Lista de usuarios a los que va dirigido la publicación o bien un string que represente grupos de usuarios, un ejemplo seria el string “TODOS” que se usaría para cuando un administrador de la plataforma quiere hacer una publicación masiva para que la vean todos los usuarios de la plataforma.

fecha_creacion(date): Fecha de publicación.
tipo(string): Dato reservado por si más adelante se categorizan las publicaciones.
estado(string): Para señalar si es publicado, eliminado, borrador, etc.

2.6 Objetivos inconclusos

En cuanto a la base de datos no hubieron objetivos inconclusos debido a que esta es de las primeras que debe estar lista para el inicio del proyecto. Pero si se podría profundizar en la seguridad de esta siguiente la siguiente checklist proporcionada por MongoDB. Cabe mencionar que varios elementos mencionados en tal checklist ya se implementan de forma automática al configurar mongo o se configuraron a través de lo realizado en esta documentación pero tales cómo desactivar la ejecución de scripts en javascript en el lado del servidor, puede que haga más falta profundizar en aquellos.

3. Sistema API: NodeJS y JSON

Antes de empezar, necesitas tener Node y mongodb instalados en el servidor, en caso de pruebas locales en el computador personal.

3.1 Raíz del proyecto: Dependencias y paquetes:

El primer código que necesitamos escribir en una aplicación basada en Node es el archivo package.json. Este archivo nos indica que dependencias vamos a utilizar en ella. Este archivo va en el directorio raíz de la aplicación:

Por lo tanto para este caso package.json tendrá lo siguiente:

```
{
  "name": "mongoose_basics",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.5.1",
    "express": "^4.16.4",
    "method-override": "^2.1.2",
    "mongoose": "^5.3.6",
    "start": "^5.1.0"
  }
}
```

Y ahora para descargar las dependencias escribimos lo siguiente en la consola o terminal usando NPM (el gestor de paquetes de Node) que se encargará de instalarlas.

```
$ npm install
```

Luego empezamos a crear directorio raíz que será el que ejecute nuestra aplicación y arranque nuestro server.

```
var express = require("express"),
    app = express(),
    bodyParser  = require("body-parser"),
    methodOverride = require("method-override");
    mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/base', function(err, res) {
      if(err) throw err;
      console.log('Connected to Database');
});

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());
```

Importamos **Express** para facilitarnos crear el servidor y realizar llamadas HTTP
Con **bodyParser** permitimos que pueda *parsear* JSON, methodOverride() nos permite implementar y personalizar métodos HTTP. Mongoose para poder hacer conexión con la DB

3.2 Creando los schemas de nuestra API (ejemplo schema de Anuncio)

```
var mongoose = require('mongoose');
var AnuncioSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    mensaje:{type: String
    },
    titulo:{type: String
    },
    autor:{type:mongoose.Schema.Types.ObjectId,
        ref:"usuario"},
    para:{type:mongoose.Schema.Types.ObjectId,
        ref:"usuario"
    },
    fecha_creacion:{type: Date ,
          default: Date.now
    },
    tipo:{type: String
    },
    estado:{type: String
    }
```

```
});
var Anuncio = mongoose.model('Anuncio',AnuncioSchema,"Anuncio");

module.exports = Anuncio;
```

3.3 Implementando los controladores de nuestras rutas:

Sección de ruteo para cada petición, get para mostrar, post agregar, put actualizar, delete borrar. Como parámetro la función del controlador correspondiente a la petición.

```
//GET - Return all anuncios in the DB
exports.findAnuncios = function(req, res) {
    Anuncio.find(function(err, anuncio) {
    if(err) res.send(500, err.message);

    console.log('GET /anuncio')
        res.status(200).jsonp(anuncio);
    });
};
```

```
//GET - Return a anuncios with specified ID
exports.findById = function(req, res) {
    Anuncio.findById(req.params._id, function(err, anuncio) {
    if(err) return res.send(500, err.message);

    console.log('GET /anuncio/' + req.params._id);
        res.status(200).jsonp(anuncio);
    });
};
```

```
//POST - Insert a new anuncios in the DB
exports.addAnuncio = function(req, res) {
    console.log('POST');
    console.log(req.body);

    var anuncio = new Anuncio({
        _id:         new mongoose.Types.ObjectId(),
        mensaje:     req.body.mensaje,
        titulo:      req.body.titulo,
        autor:       req.body.autor,
        para:        req.body.para,
        tipo:        req.body.tipo,
        estado:     req.body.estado
    });

    anuncio.save(function(err, anuncio) {
```

```
        if(err) return res.send(500, err.message);
    res.status(200).jsonp(anuncio);
    });
};
```

```
//PUT - Update a register already exists
exports.updateAnuncio = function(req, res) {
    Anuncio.findById(req.params._id, function(err, anuncio) {
        anuncio.mensaje=    req.body.mensaje,
        anuncio.titulo=     req.body.titulo,
        anuncio.autor=      req.body.autor,
        anuncio.para=       req.body.para,
        anuncio.tipo=       req.body.tipo,
        anuncio.estado=    req.body.estado

        anuncio.save(function(err) {
            if(err) return res.send(500, err.message);
      res.status(200).jsonp(anuncio);
        });
    });
};
```

```
//DELETE - Delete a anuncios with specified ID
exports.deleteAnuncio = function(req, res) {
    Anuncio.findById(req.params._id, function(err, anuncio) {
        anuncio.remove(function(err) {
            if(err) return res.send(500, err.message);
      res.status(200).send("borrado");
        })
    });
};
```

Funciones para buscar, agregar, actualizar o eliminar documentos de la bd correspondientes a la colección que se este usando. Esto por medio de la ruta de las rutas correspondientes(get/post/put/delete).

tenemos que unir estas funciones a las peticiones que serán nuestras llamadas al API. Volvemos a nuestro archivo principal, el directorio raíz y declaramos las rutas.

```
app.use(router);

// API routes tecnica
var tecnica = express.Router();

tecnica.route('/tecnica')
  .get(tecnicaCtrl.findTecnicas)
```

```
  .post(tecnicaCtrl.addTecnica);

tecnica.route('/tecnica/:_id')
  .get(tecnicaCtrl.findById)
  .put(tecnicaCtrl.updateTecnica)
  .delete(tecnicaCtrl.deleteTecnica);

app.use('/', tecnica);
```

3.4 Complejidades y notas a considerar:

Lo que fue más difícil y demoró más tiempo fue crear los schemas, para que sean igual a las de la base de datos y luego buscar como hacer las rutas, para que el front-end se pueda conectar a la base de datos mediante la api.

3.5 Puntos faltantes y observaciones:

- Falta validación de los datos de entrada:
    - Corroborar datos obligatorios para la creación de los objetos JSON.
- Seguridad en general:
    - Pruebas de estrés, caídas de sistema y servidor.
- Restricciones y limitación de acceso:
    - Evaluar la implementación de tokens.
    - Revisar privilegios para usuarios, módulos y cursos.
- Encriptación de los datos enviados y recibidos.
- Falta implementar ciertas rutas:
    - sesion_de_estudio: sesion, técnicas, tipos.
    - anuncios: usuarios.
- Función DELETE:
    - Restringir uso para los usuarios.
    - Usar “etiqueta” booleana para definir al usuario eliminado.

4. Diseño plataforma:

Módulos del frontend que no fueron asignados a algún grupos del fronted:

- Página de aterrizaje: aspectos como minimalista en cuanto a colores (tonos blanco y gris), logo de la Universidad y del colegio de ayudantes. Mantener estable comunicación con María Constanza Uribe para el diseño.
- Administración de plataforma: Dashboard para el superusuario, monitoreo de los accesos y registros de participantes o grupos. Mantener estable comunicación con Beatriz Moya.