Universidad Autónoma de Querétaro

Facultad de informática

Manual Técnico: Sistema Para Seguimiento a la Carrera Docente

Descripción del código fuente, estructura de carpetas, y tecnologías.

Escrito por:

Daniel Sierra Reveles

sd.reveles@gmail.com

Centro de Desarrollo

Table of Contents

Leguajes Y Tecnologías Utilizados 3
TypeScript 3
NodeJS 3
MongoDB 3
GraphQL 3
PDFtk server 3
Configuración de NodeJS 3
Dependencias utilizadas (package.json) 3
Variables de entorno (.env) 4
Constantes globales (config.const.ts) 4
Configuración servidor y endpoints 4
TypeScript 6
GraphQL 6
Schema 6
- Definición de tipos 6
- Queries, mutations y suscriptions. 7
- Index del schema: 7

Resolvers 8
MongoDB 10
Utils 11
IsAuth 11
logAction 11
categoriesHelper 11
Merge 11
imageUploader 11
REST endpoints 12
Estructura De Proyecto 12
Instalación 13

Leguajes Y Tecnologías Utilizados

TypeScript

Lenguaje de programación desarrollado y mantenido por Microsoft que funciona como un superset de JavaScript (JS) dándole a este las capacidades de funcionar como un lenguaje orientado a objetos y provee de un tipado estático opcional. Funciona trans-compilando el código escrito en TypeScript a código JS en la versión especificada en el archivo de configuración. No remplaza a JavaScript en su totalidad ya que código de JS es completamente funciona dentro de TypeScript.

NodeJS

Entorno de ejecución de javascript para el desarrollo de backend, tras bambalinas ejecuta código de c++ para realizar la locación de objetos en memoria por lo que posee un buen rendimiento. Un punto fuerte de nodejs es su manejo de websockets y buffers los cuales son normalmente utilizados en la forma de streams para la manipulación de datos dentro de un flujo en memoria sin necesidad de almacenarlos temporalmente en disco y sin ocupar gran cantidad de memoria.

MongoDB

Base de datos NoSQL basado en documentos, Sus principales diferencias de las bases de datos relaciones es que no utiliza SQL para las consultas e inserciones. Utiliza colecciones en lugar de tablas y documentos en lugar de filas, no posee un esquema fijo para las colecciones, sino que cada documento dentro de una colección puede tener la forma que el programador crea conveniente, también es posible que los documentos contengan subdocumentos o arreglos de documentos como campos.

GraphQL

Es un lenguaje para consultas de API, funciona hasta cierta medida como alternativa a APIs REST diferenciándose de este en los siguientes puntos:

- Dentro de la consulta el desarrollador frontend especifica que campos son los que desea recibir únicamente del API evitando el over-fetching y el under-fetching
- El API de graphql solo es capaz de enviar JSON
- Estructurado de manera correcta permite obtener información en una sola consulta que en un API REST requeriría múltiples consultas
- Integra un tipado estático a las entidades involucradas en modelo de negocios.

PDFtk server

Es una herramienta de consola que se encarga de la manipulación de archivos PDF. Dentro de este proyecto se utilizó para realizar merge de archivos pdf, no es necesario que el usuario interactúe con la aplicación directamente, su funcionalidad se encuentra encapsulado en un binario escrito en go. **El código fuente de este binario se encuentra en la carpeta PDFMerger**.

Configuración de NodeJS

Dependencias utilizadas (package.json)

Archivo de configuración principal de un proyecto de nodejs, en el se pueden encontrar las dependencias, o librerías externas usadas en el proyecto. Considerando que todas las librerías son necesarias podemos dividirlas en las que dan estructura general al proyecto y las que son de soporte o cumplen con alguna función en específico.

- Las librerías de estructuras son:
  - Apollo-server
  - Mongodb
  - Mongoose
- Las librerías de soporte son:
  - Archiver
  - Cors
  - Dotenv
  - Jsonwebtoken
  - Promise-all

Variables de entorno (.env)

Este archivo funciona para almacenar variables de entorno que estarán disponibles desde cualquier parte de la aplicación.
El archivo se compone de pares key-value, considerando todo como string, no es necesario colocar comillas dobles ni simples Como convención los nombres de las variables van en mayúsculas todas sus letras por tratarse de constantes.
Para acceder a las variables desde la aplicación se utiliza el objeto global de nodejs proccess.env.VARNAME.

Constantes globales (config.const.ts)

Sirve para almacenar constantes de la aplicación pero que no son parte de la configuración de la aplicación.

Configuración servidor y endpoints

El archivo server.ts es donde se configuran las librerías que en conjunto darán funcionalidad al servidor. A continuación, se describe cada parte de este archivo.
Imports: Consiste en la importación de librerías y archivos del proyecto necesario para configurar el servidor.

```
1   import { typeDefs } from './graphql/schemas/index';
2   import { resolvers } from './graphql/resolvers/index';
3   require('dotenv').config()
4   import { getUser } from "./utils/is-auth";
5   const mongoose = require('mongoose');
6   const { ApolloServer } = require('apollo-server-express');
7   const express = require('express');
8   const bodyParser = require('body-parser');
9   const cors = require('cors');
10  const port = process.env.PORT || 4000;
11  import { router } from './routes/downloads.route';
```

Configuración mongoose: Se utiliza las variables de entorno que contiene los datos necesarios para la conexión a la db y se crea un objeto de conexión que servirá por mongoose para realizar todas las consultas posteriormente.

```
13  mongoose.connect(
14    process.env.DB_PATH + '/' + process.env.DB_NAME,
15  > {···
20    }
21  );
22
23  const db = mongoose.connection;
24  db.on('error', console.error.bind(console, 'connection error:'));
25  db.once('open', () => {
26    console.log('Connected to the database');
27  });
```

Configuración de Apollo: Apollo es la librería utilizada para implementar graphql y requiere en su configuración los siguientes datos:

- **typeDefs**: Es el objeto gql generado a partir del schema.
- **Resolvers**: Es el objeto que en sus propiedades contiene todas las funciones que dan lógica al schema.
- **Instrospection** y **playground** son opciones que son sumamente útiles durante el desarrollo del sistema ya que permiten acceder y consultar el API de manera gráfica, sin embargo, se recomienda desactivarlas en un entorno de producción.
- **Context**: dentro de este campo se ejecuta una función anónima la cual obtiene como parámetro los metadatos del request recibido por http, Al final de esta función se puede regresar un objeto el cual estará disponible para todos lo resolvers en el parámetro context, es aquí donde se pueden utilizar middlewares o agregar información al request.

```
33  // apollo server
34  const server = new ApolloServer({
35    typeDefs,
36    resolvers: resolvers,
37    introspection: true,
38    playground: true,
39    context: ({ req }) => {
40      // get the user token from the headers
41      const token = req.headers.authorization || '';
42      // try to retrieve a user with the token
43      const user = getUser(token);
44      user['ip'] = req.ip;
45      // add the user to the context
46      return { user };
47    },
48  });
```

Configuración de servidor de express: Debajo de Apollo se encuentra un servidor en el framework express, el cual primero se instancia y después se comienzan a agregar middlewares y routers o endpoints, siendo graphql a nivel express un endpoint más. Por último una vez configurado el servidor se inicializa con el comando listen y el servidor comienza a escuchar y responder las peticiones.

```
29  const app = express();
30  app.use(cors());
31  app.use(bodyParser.json());
32
33  // apollo server
34  const server = new ApolloServer({···
48  });
49  server.applyMiddleware({ app, path: '/graphql' });
50  // /public only is used when the images files are stored in
51  app.use('/public', express.static(__dirname + '/public'));
52  app.use('/downloads', router);
53
54  app.listen({port}, () => {
55    console.log(`🚀  server ready at http://localhost:${port}${server.graphqlPath}`);
56  });
57
```

TypeScript

En este proyecto TypeScript se encuentra como dependencia de desarrollador, pero puede ser instalado globalmente con el comando `npm install -g typescript` una vez instalado es posible acceder a los comandos de consola. Para compilar el proyecto es necesario estar en la raíz del mismo (donde se encuentra el archivo tsconfig.json) y ejecutar el comando `tsc` con esto el compilador generara el código en JS en la carpeta dist. Para un entorno de desarrollo en el que se desee una compilación automática y más ágil, existe el comando `tsc -w` el cual dejara el compilador en modo de vigila y automáticamente compilara los archivos donde se detecten cambios

GraphQL

Un proyecto de graphql se compone de 2 partes principales, el schema y los resolvers, y de 3 diferentes maneras para identificar el tipo de consultas que se desean realizar, query, mutation y subscription.

Un query es aquel que únicamente consulta información, pero no la modifica, las mutations se encargan de la creación, modificación y eliminación, mientras que las subscription mantienen un socket abierto para que existe un canal en tiempo real de ese query en específico.

Schema

es la definición de los tipos de datos, los queries, mutations y subscripciones que existen en la aplicación, cualquier función no definida en el schema es ignorada por graphql.

- Definición de tipos

```
"""
Status:
0.- "Oculto"
1.- "Vigente"
2.- "Vencido"
"""
input UpdateNotice {
  title: String
  body: String
  status: Int
  link: String
  fromDate: Float
  toDate: Float
}
"""
Activo
Inactivo
"""
type User {
  _id: ID!
  clave: String!
  status: String!
  name: String!
  lastName: String!
  adscription: Campus!
  permissions: [Permission!]!
}
```

Comentarios en el schema

  - Subrayado en morado: Existen 2 posibilidades, *input* para variables de entrada y *type* para variables de salida

- Subrayado en amarillo: Nombre del objeto de tipo de dato,
- Subrayado en naranja: Nombre de cada una de las propiedades (o campos)
- Subrayado en gris: El tipo de escalar o un tipo de dato previamente definidor en el esquema que se asigna a la propiedad, Se coloca un signo de admiración para definir que esta propiedad no puede regresar un valor nulo en los objetos de tipo *type* y que es una propiedad obligatoria en los objetos de tipo *input*. Para definir que el campo es una lista se colocan square brackets alrededor del escalar.

- Queries, mutations y suscriptions.

```
25  createUser(input: InputUser!): User!
26  """
27  UpdateUser:                          } Comentarios
28  status: "Activo" || "Inactivo"       }
29  """
30  updateUser(id: ID! status: String!): User!
31  updateUserRole(input: UpdateUserRole!): User!
32  deleteUser(id: ID!): User!
```

- Subrayado de verde: Nombre de la consulta, queries, mutations y subscriptions comparten la misma estructura. Es importante resaltar que el nombre definido en el schema debe de coincidir con el usado en la función que dará lógica como resolver.
- Subrayado en rosa: Parámetros de entrada de la consulta, Se pueden utilizar únicamente escalares o tipos de dato de tipo *input* definidos en el esquema.
- Subrayado en naranja: El tipo de dato que regresara al cliente que realice la consulta. Solo pueden ser escalares o tipos de dato de tipo *Type*.

- Index del schema:
- Todas las reglas que dan la estructura al schema son utilizadas dentro de un string, este string es después convertido en un objeto de tipo 'gql' que la librería de graphql puede interpretar para aplicar todas estas reglas.
  - Para la creación de este objeto gql se hace uso de template literals para la unión de los distintos strings, los cuales son importados de los archivos correspondiente a cada tipo de consulta.

```
1   import { gql } from 'apollo-server';
2   import { types } from './types';
3   import { queries } from './queries';
4   import { mutations } from './mutations';
5
6   export const typeDefs = gql`
7     ${types}
8
9     type Query {
10      ${queries}
11    }
12
13    type Mutation {
14      ${mutations}
15    }
16  `;
```

Resolvers

Es el código que da la lógica a cada uno de los queries, mutations y subscriptions declarados. La manera en que se organizaron en este proyecto fue que se implementó un objeto por cada tipo de consulta, por cada modelo existente en las reglas de negocio, dentro de cada uno de estos objetos como propiedades incluye cada uno de los queries, mutations o subscriptions definidos en el schema, si fue definido en el schema y no existe en los resolvers como función el servidor arrojara un error de falta de implementación del schema.

```
59    /**
60     * Get multiple categories
61     * @param {number} args.page - number of page for pagination
62     * @param {number} args.perPage - number of elementes to show in each page
63     * @param {number} args.type - type of category, root, leaves or all
64     * @return { [Category] } - a list of category documents
65     */
66    categories: async (_, args, context, info) => {
67      const qType = 'Query';
68      const qName = 'categories';
69      try {
70        if (!await isAuth(context, [config.permission.docente])) {···
73        }
74
75        const projections = getProjection(info);
76        let conditions;
77        if (args.type === 1) {···
79        } else if (args.type === 2) {···
81        } else if (args.type === 3) {···
83        }
84        const docs = await Category.find(conditions, projections, {sort: {clave: 1}}).exec();
85        if (projections.children) {···
87        }
88        registerGoodLog(context, qType, qName, 'Multiple documents');
89        return docs;
90      } catch (e) {
91        registerErrorLog(context, qType, qName, e);
92        throw new ApolloError(e);
93      }
94    },
```

Ejemplo de un resolver. Se puede apreciar que en esencia es una función asíncrona de javascript aunque posee algunas características propias de un resolver como los parámetros que recibe (parent, args, context, info) esta función es llamada por el servidor automáticamente al recibir la petición por parte del cliente y es el servidor quien provee de estos parámetros automáticamente. El resto de la función se compone de autenticación, lógica de negocio, registro de eventos, manejo de errores y por ultimo la el regreso del resultado obtenido hacia el cliente.

A continuación, se describen los elementos no relacionados con la lógica que se esta manejando en este resolver en específico.

- _, args, context, info: son los 4 parámetros que un resolver posee por default,
  - El primero es el resolver padre de donde se esta ejecutando, en este proyecto no se utilizo el anidado de resolvers por lo que este parámetro siempre se encuentra omitido haciendo uso de un guion bajo.
  - **Args** contiene los argumentos o parametros con que fueron pasadons junto con la consulta, estos parametros tienen el mismo nombre con el que fueron definidos en el schema.

- **Context** contiene la información que definamos en la configuración del servidor y sirve para contener información provista por middlewares, es el lugar ideal para almacenar la información del usuario que realiza la consulta
- **Info** contiene todos los metadatos de la consulta, la información almacenada en este parámetro es demasiada pero dentro de este proyecto se utiliza para obtener que campos son los que el usuario esta solicitando en la consulta y de esta manera obtener únicamente esos mismos campos de la base de datos y no solicitar datos que no será utilizados. Para realizar esta acción se utiliza una función auxiliar que lee el objeto info y regresa un objeto listo para ser consumido por mongoose como projections (selects).

- La autenticación en graphql puede ser manejada de 2 maneras, aplicada a absolutamente todas las consultas o individualmente. Debido a las reglas de negocio de este proyecto se optó por ser integradas individualmente en cada resolver, para ello las primeras líneas de código dentro de la función son la verificación de permisos haciendo uso de una función auxiliar que obtiene la información almacenada en el context y una lista de los permisos que debe de tener el usuario para acceder a el resolver en cuestión, los permisos se encuentran definidos en el archivo config.const.ts que es un enum para hacer humanamente leíble los permisos.
- Qtype, qName y registerXlog: Son variables y funciones que se encargan de construir un registro con toda la información de la consulta para posteriormente ser almacenada en la base de datos.
- El comentario previo a cada resolver es la documentación básica de la función escrita en el estilo de JSDoc.

Al igual que el schema todos los objetos que definen los resolvers se encuentran dispersos para su mejor organización, pero deben de ser unificados para consumo que el servidor pueda hacer uso de ellos. En el caso de los resolvers en lugar de utilizar **template literals** se utiliza **destructuring assignment** ya que en esta ocasión no estamos trabajando con strings sino con objetos de javascript.

```
6   import { systemLogQueries, systemLogMutations } from './systemLog.resolver';
7   import { userQueries, userMutations } from './user.resolver';
8   import { uploadsMutations, uploadsQueries } from './uploads.resolver';
9   import { loginQueries } from './authLogin.resolver';
10
11  export const resolvers = {
12    Query: {
13      ...documentQueries,
14      ...campusQueries,
15      ...categoryQueries,
16      ...noticeQueries,
17      ...permissionQueries,
18      ...systemLogQueries,
19      ...userQueries,
20      ...uploadsQueries,
21      ...loginQueries
22    },
23    Mutation: {
24      ...documentMutations,
25      ...campusMutations,
26      ...categoryMutations,
27      ...noticeMutations,
28      ...permissionMutations,
29      ...systemLogMutations,
30      ...userMutations,
31      ...uploadsMutations
32    }
33  };
```

MongoDB

Para el uso de mongodb en este proyecto se utilizaron 2 librerías, la oficial de mongodb para nodejs y mongoose.
La librería oficial fue utilizada para el almacenamiento, recuperación y manipulación de los archivos pdf en la base de datos, por medio de GridFS, En el archivo *download.routes* en la carpeta *routes* se ve este uso en las funciones *getConnection()*, *getGrid()* y la manipulación de los buckets de GridFS por medio de streams.
La librería mongoose es la que se encarga de hacer las interacciones de todas las demás colecciones. Mongoose utiliza modelos a los que se les define una estructura y que colección es la que afectara. Sobre estos modelos creados por mongoose se ejecutan todo el CRUD con los diferentes métodos provistos por la librería.

Ejemplo de modelo:

src > models > TS category.model.ts > [⊙] categorySchema

```
1   const mongoose = require('mongoose');
2   const Schema = mongoose.Schema;
3
4   const categorySchema = new Schema({
5     root: {type: Boolean, required: true },
6     clave: {type: String, required: true},
7     title: {type: String, required: true},
8     path: {type: String, required: true},
9     value: {type: Number},
10    children: [
11      {type: Schema.Types.ObjectId, ref: 'Category'}
12    ]
13  });
14
15  export const Category = mongoose.model('Category', categorySchema, 'Categories');
16
```

- Subrayado en morado: nombre del campo.
- Subrayado en verde: tipo de dato que guardara.
- Subrayado en amarillo: reglas y opciones para el campo.
- Subrayado en naranja: Nombre del modelo.
- Subrayado en rojo: Nombre de la colección.

```
109
110       const tempCat: Category = await CatModel.findById(
111         category, {_id: true, clave: true, title: true, value: true, children: true});
112
```

- Subrayado en blanco: datos que se obtiene de la ejecución de la consulta.
- Subrayado en amarillo: modelo de mongoose
- Subrayado en rojo: método de consulta
- Subrayado en verde: criterios de búsqueda, dependiendo del método de consulta puede ser un id o un objeto de criterios donde se especifica que es lo que se esta buscando, equivalente a WHERE en SQL
- Subrayado en rosa: campos que se desean obtener del documento buscado, equivalente a SELECT en SQL

Utils

Son clases y funciones que tiene como objetivo la restructuración y reutilización del código, El uso de estas funciones se encuentra por todo el sistema por lo que no forman parte de ningún área en especifico y necesitan estar en un espacio propio para su organización.

IsAuth

En este archivo se encuentran 2 funciones, getUser y isAuth.

getUser se ejecuta en cada request dentro del context y rutas REST, y su objetivo es decodificar el jwt, extraer el usuario que está realizando la petición y colocar el userId dentro del objeto de context o en caso de que se no provea un token o que esté sea invalido se regresara Unauthenticated como userId y un false como autenticado.

isAuth se encarga de saber si un usuario posee los permisos que se están solicitando, en caso de tenerlos se regresa un true y un false en caso contrario.

logAction

El objetivo de estas funciones es proveer al desarrollador de una manera clara y sencilla de llevar un registro del sistema claro y consultable.

categoriesHelper

La entidad categorías posee queries un tanto específicos los cuales de implementar toda la lógica dentro de los resolvers los convertiría en un claro ejemplo de code smell y haría de su mantenimiento algo innecesariamente complicado. Es por eso que en este archivo se refactorizo muchas de las funciones y se definen tipos de datos específicos para manejar de manera más clara la información que se maneja en estas funciones.

Merge

Existen múltiples ocasiones en las que un documento posee una propiedad que almacena el id de otro documento, posiblemente de un documento de otra colección, sin embargo, el usuario no le sirve obtener únicamente este id, sino que necesita el documento al que apunta. Para poder regresar el documento hijo dentro de la propiedad del documento padre es necesario recuperarlo de la base de datos y para esto existen múltiples funciones que realizan esta acción, reciben un documento con ids y regresan un documento con los documentos ya unidos en sus propiedades.

Estas funciones se fueron creado bajo demanda, pero la premisa es la misma, recibe un objeto con ids, ejecuta la búsqueda de dicho documento en la base de datos y sustituye el id con el documento completo.

Estas funciones también permiten que esta acción sea realizada únicamente cuando el cliente lo solicite, no se realizan consultas a la db a menos de que el cliente haya solicitado ese campo en específico, esto incluye situaciones donde se presente dependencia circular entre documentos.

imageUploader

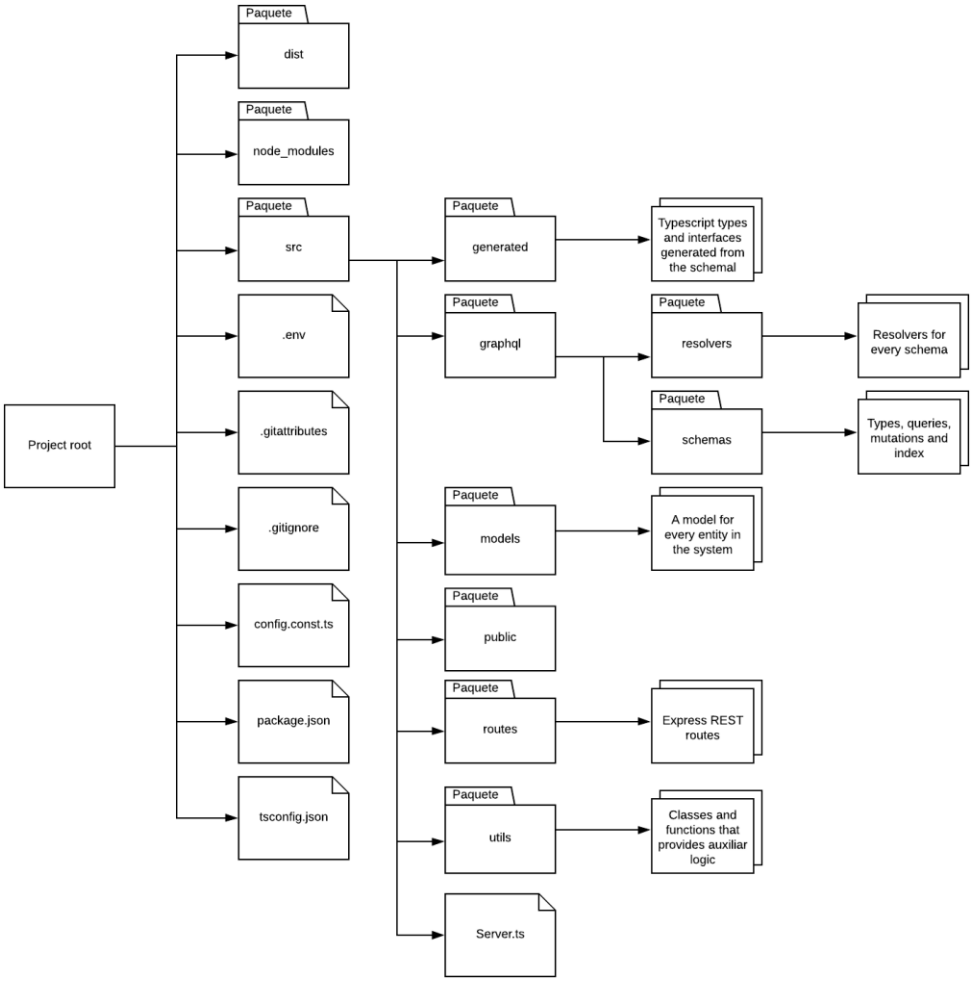
Funciones para el almacenamiento de imágenes tanto en el bucket de S3 como en el sistema local.

REST endpoints

El sistema no se encuentra expuesto únicamente por graphql, dada la limitación de graphql de poder únicamente transmitir JSON existen 3 endpoints para la obtención de los pdf, `/getFile, /joinInZip y /joinInPdf.` Se encuentran disponibles como subrutas de /downloads y los 3 son HTTP POST. Su implementación esta dentro de la carpeta routes en el archivo downloads.route.ts, ahí mismo se encuentra documentado los parámetros que cada uno de estos recibe para ser utilizados.

Estructura De Proyecto

Carpeta dist: Contiene el compilador de typescript a javascript, dentro de esta carpeta se encuentran los archivos que se utilizaran para correr el servidor en nodejs. No modificar.
Carpeta node_modules: Contiene las librerias de terceros necesarias para que el sistema funcione. No modificar.
Carpeta src: Contiene el código fuente en typescript de todo el sistema, estos son los archivos que se deben de modificar en caso de requerir algún cambio.
.env : Archivos donde se definen las variables de entorno



Instalación

1. Clonar el repositorio
2. Configurar las variables de entorno, se pueden configurar usando un archivo llamado “.env” en la raíz del proyecto.
    a. Las variables de entorno necesarias son las siguientes:

```
DB_PATH=mongodb://[user name]:[user pwd]@[db address]:[db port]
DB_NAME=[db name]
DB_AUTH_COL=[name of the collection that provides auth to the db]
PRIVATE_KEY=[private key for jwt generation]
API_LOGIN=https://portalinformatica.uaq.mx/portalInformatica/portal-informatica-api-iniciar-sesion

ANONYMOUS_URL=[url for default user profile pic]
INFO_CAMPUS_ID=[campus id ]

BUCKET_NAME=[S3 amazo bucket name]
IAM_USER_KEY=[S3 credentials]
IAM_USER_SECRET=[S3 credentials]

```

3. Ejecutar `npm install` desde la raíz del proyecto
4. Ejecutar el archivo javascript de servidor en nodejs, dicho archivo se encuentra en dist/src/server.js. Desde la raíz del proyecto se ejecuta el comando node dist/src/server.js
5. Si el servidor y base de datos se encuentran configurados correctamente en las variables de entorno el servidor deberá mostrar el siguiente mensaje:

Modificar el código

Cuando se presente la necesidad de modificar el código se recomienda no modificar la carpeta dist directamente, sino en su lugar modificar el código en typescript y recompilar.