

Desarrollo de Aplicaciones Web (DAW)

Alumno Daniel Marín Egea

Tutor Lorenzo Gonzalez Gascon



CIPFP Mislata

Centre Integrat Públic
Formació Professional Superior

Licencia

[Attribution 4.0 International](<https://creativecommons.org/licenses/by/4.0/>)

[Legal Code](<https://creativecommons.org/licenses/by/4.0/legalcode>)

Usted es libre de:

Compartir:

Copiar y redistribuir el material en cualquier medio o formato

Adaptar:

Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.



Índice

1. Marco de la investigación	1
1.1. Temática elegida	
1.2. Contextualización	
2. Organización del proyecto	3
2.1 Temporalización	
2.2. Recursos	
3. Aplicación práctica	7
3.1 Introducción	
3.2. Ciclo de vida	
3.2.1 Análisis de requerimientos	
3.2.2 Diseño	
3.2.3 Implementación	
3.2.4 Pruebas	
4. Manual de uso	35
5. Valoración personal del proyecto	42
6. Fuentes bibliográficas	43

1. Marco de la investigación

1.1. Temática elegida

La idea de este proyecto se basa en que los usuarios se creen su propio “blog” y puedan publicar artículos de manera sencilla, así las demás personas puedan visualizar los artículos publicados, con una interfaz y navegación simple.

El objetivo de este proyecto de investigación será basado en en peticiones al servidor mediante consultas json con la tecnología llamada **GraphQL**.

Ejemplo petición GraphQL

```
{
  posts {
    title
    description
    content
    author {
      username
      email
    }
  }
}
```

El proyecto se divide en dos partes, en backend y en frontend ambas partes se realizarán utilizando el lenguaje de programación Javascript que es un lenguaje de programación no tipado, ligero e interpretado.

Javascript es un lenguaje que se ejecuta en el navegador, para poder realizar la parte de servidor con Javascript es necesario utilizar un entorno donde se ejecuta código Javascript en la parte del servidor para ello será necesario NodeJS y para realizar peticiones a un servidor usaremos **Express** y finalmente para poder realizar peticiones GraphQL usaremos **Apollo Server**, ambas son librerías para poder realizar estas funcionalidades.

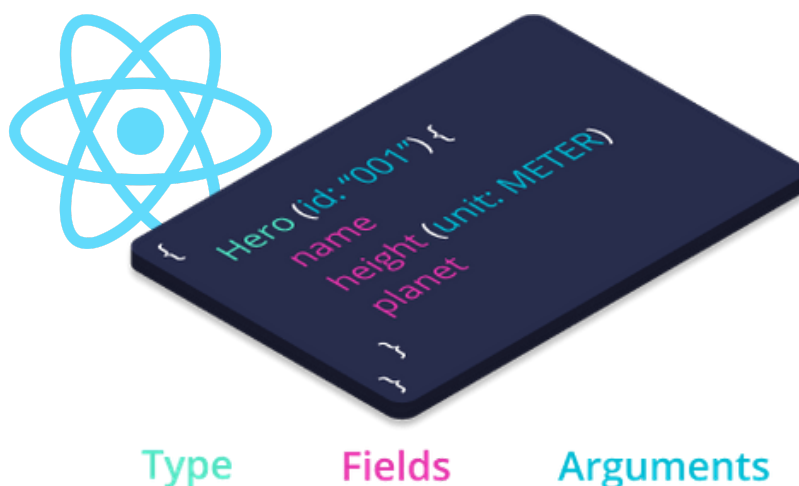
Para el entorno del lado cliente usaremos React es una librería/framework que ayuda a construir interfaces para el usuario, para poder realizar peticiones al servidor GraphQL utilizaremos **Apollo Client** (React).

1.1. Contextualización

Se realizará una investigación e elaboración del proyecto basado en GraphQL ya que es una forma distinta a la **REST API** común, con GraphQL podremos realizar peticiones al servidor desde una única dirección, simplemente enviando una query te devuelve los datos pedidos de la query. GraphQL no solo se queda ahí sino que también es una buena forma de tipar los datos y las peticiones al servidor.

Del lado cliente elegí realizar la investigación de React una librería de Javascript, ya que es una librería bastante ligera donde puedes configurar totalmente el entorno de desarrollo o simplemente utilizar una configuración por defecto que te otorga **Create React App** pero aún así puedes configurar todo el entorno, con un simple comando te reestructura todo el proyecto para la configuración.

React es uno de los frameworks más utilizados junto con **Angular** y **VueJS**, quise realizar esta investigación ya que por un lado Angular fue visto e impartido durante el curso, además de que mi proyecto de investigación se basa en **GraphQL** y tanto React como GraphQL están desarrollados por la misma empresa (facebook).



2. Organización del proyecto

2.1. Temporalización

Lo primero era cuestionar qué tecnología y lenguaje de programación utilizará, al final me decante por **Javascript** ya que era un lenguaje que se iba a utilizar para la parte de cliente y me parecía interesante la utilización del lenguaje en entorno servidor ejecutándose mediante **NodeJS**.

Para empezar a realizar este proyecto era necesario la investigación de GraphQL en un servidor ejecutándose código javascript, anteriormente hice una prueba de cómo funcionaba GraphQL en Laravel (PHP), pero realizarlo en javascript era totalmente distinto.

Empecé a indagar cómo podría empezar un proyecto de GraphQL en javascript, habían varias opciones, pero al final me decante por Apollo, ya que tenía 2 librerías para poder realizar tanto el servidor como el cliente en React.

Al principio para realizar los esquemas y resolvers de GraphQL utilice la librería **apollo-boost** ya que te otorga una configuración por defecto, pero que al final tuve que utilizar otras librerías para definir la configuración del servidor y los esquemas ya que la configuración básica no me permitía la subida de archivos.

Para empezar a realizar este proyecto era necesario la investigación de React, ya que no tenía conocimientos previos al realizar este proyecto, tuve que indagar cómo empezar un proyecto de React ya que no era un framework en sí mismo, sino una librería.

La preparación del entorno React fue tan clara cuando encontré **Create React App** un entorno configurado por defecto hecho en **Webpack** te permite escribir código modular y empaquetarlo junto en paquetes más pequeños que optimizan el tiempo de carga y **Babel** como compilador, este compilador te permite escribir javascript más moderno y que aún así funcione en navegadores más antiguos.

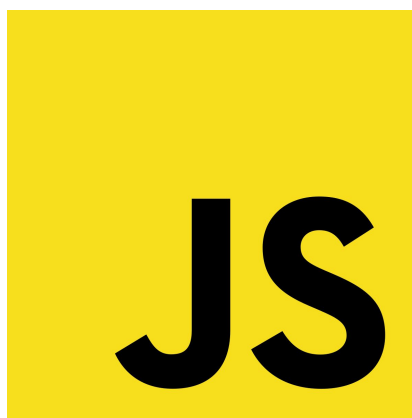
Una vez investigado GraphQL y React lo siguiente ya era darle la lógica al proyecto y organizarlo para saber qué estructura y diseño debería de tener.

2.2. Recursos

Lenguajes:

Como mencione anteriormente en este proyecto la idea general era utilizar un solo lenguaje para ambas partes, para backend y para frontend.

El lenguaje de programación **Javascript** es actualmente es uno de los más utilizados, además de que con este lenguaje no solo se dedica en algo en específico sino que gracias a ciertas librerías es posible la utilización en otros entornos.



Librerías:

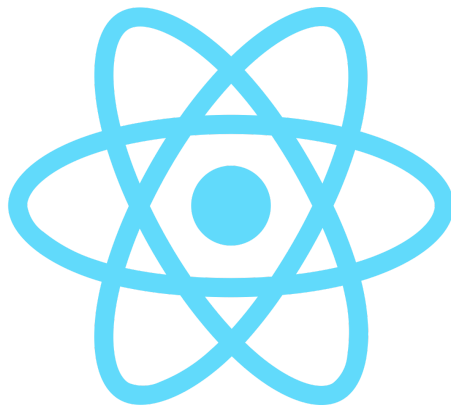
La librería por excelencia del proyecto es **Apollo** tanto para servidor como para cliente ya que te permite la utilización de **GraphQL** en servidor para definir los esquemas y los resolvers como para mandarle peticiones desde el cliente.



NodeJS es totalmente necesario en el lado de servidor para poder realizar la ejecución del lenguaje de programación javascript en el entorno servidor



React para la construcción de una interfaz para el usuario en el lado del cliente

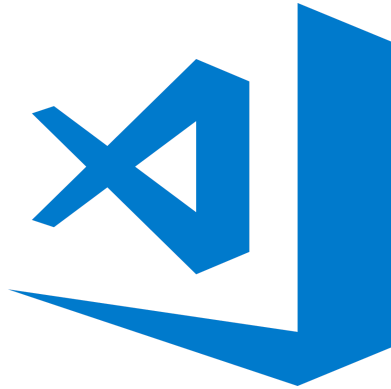


Tailwind CSS para los estilos y diseño de la página

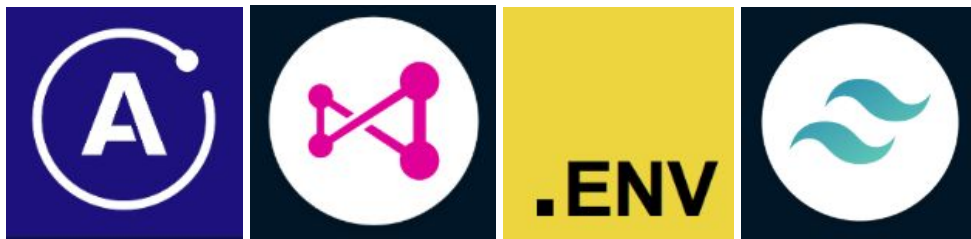


Herramientas:

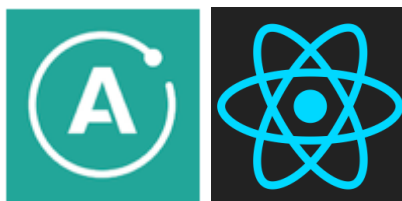
Visual Studio Code para programar el proyecto



Extensiones para Visual Studio Code **GraphQL**, **Apollo GraphQL**, **DotENV**, **Tailwind CSS** y **IntelliSense**, para una mejor comodidad al programar.



Extensiones para Chrome **Apollo Client Developer Tools** y **React Developer Tools** para debuggear en el navegador.



3. Aplicación práctica

3.1. Introducción

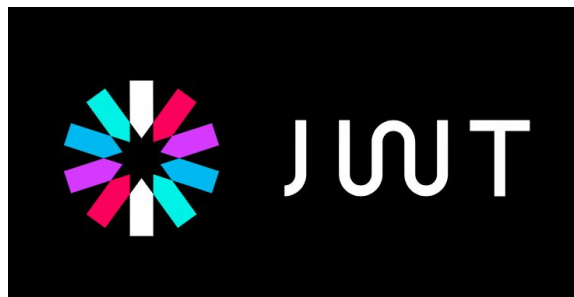
Para la creación de este proyecto es necesario seguir unas pautas y esquemas para organizar el proyecto, ya que tanto **Apollo** y **React** son librerías no tienen una estructura de carpetas estándar definida o la definición del proyecto o estructura de la base de datos, en este apartado daré los detalles de cómo el proyecto a ido tomando forma conforme avanzaba.

3.2. Ciclo de vida

3.2.1 Análisis de requerimientos

El uso básico de la aplicación no requiere requerimientos especiales, únicamente los únicos requerimientos que pide la aplicación son de **autenticación JWT** para la gestión de la creación de blogs y publicación de artículos.

Además hay un requisito especial añadido para la producción de un blog, el requerimiento es la verificación de email a no ser que provenga de un registro OAuth, simplemente se le enviará al usuario un email con una dirección url que verificará al usuario para la creación de un blog.

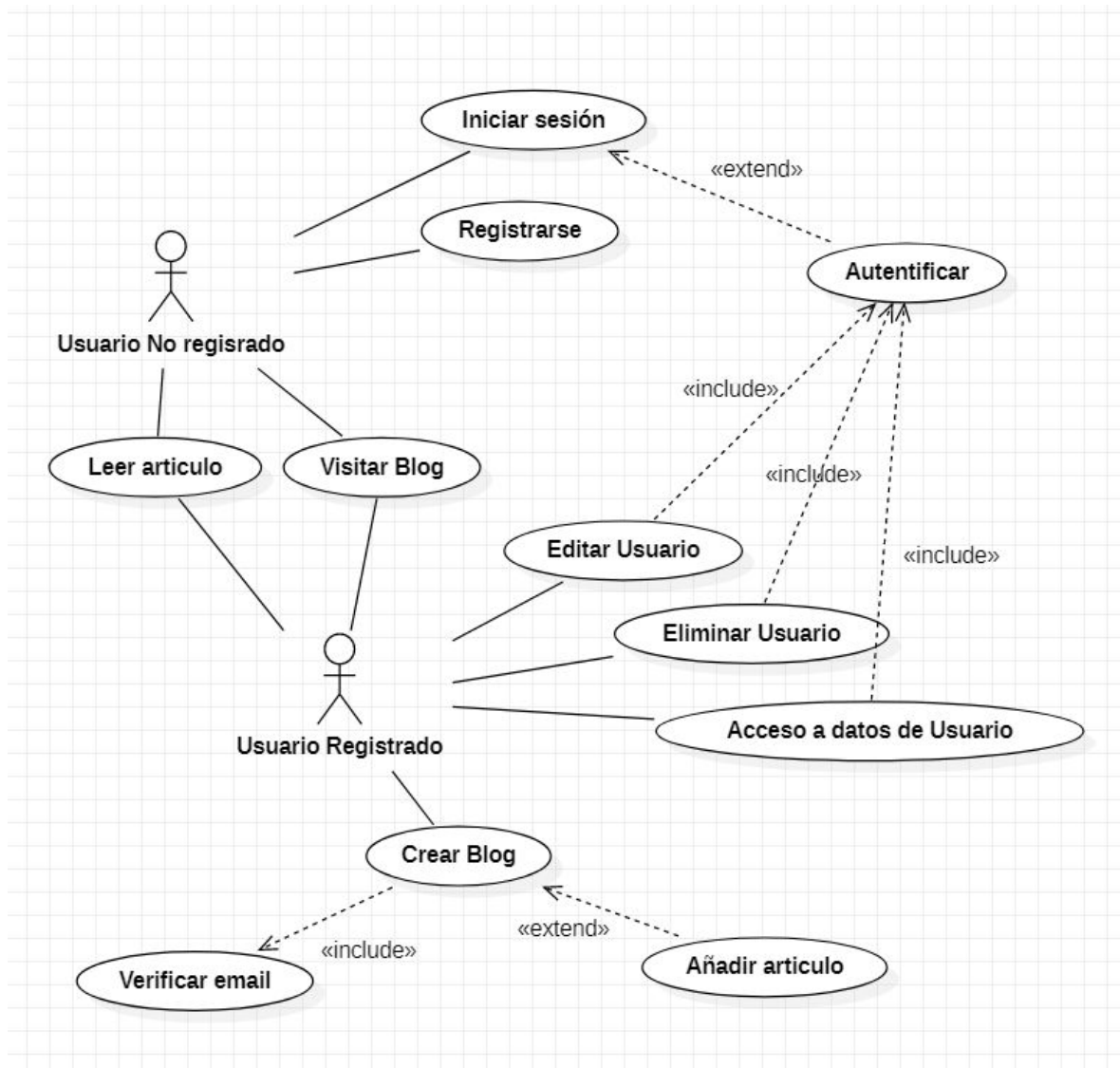


3.2.1 Diseño

Casos de uso

Con el diagrama de casos de uso definiremos como funcionara la aplicación y cómo podrá interactuar el usuario dentro de la aplicación.

Diagrama de casos de uso



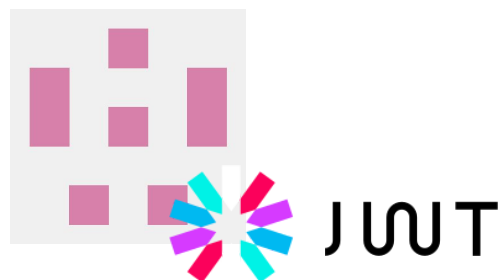
Acceso

Definición de las acciones que podrían acceder los usuarios en la aplicación según su estado.

Acceso de los usuarios tanto autenticados como invitados en la aplicación.

Identificadores	Descripciones de acceso
Usuarios (<i>No registrados</i>) [uNO]	<ul style="list-style-type: none">- Visitar Blogs- Leer artículos publicados por los usuarios registrados- Autenticación- Registrarse- Registrarse mediante OAuth (google, github)
Usuarios (<i>Registrados</i>) [uR]	<ul style="list-style-type: none">- Mismo acceso que los <i>usuarios (No registrados)</i>- Modificación de datos- Descarga de información guardada del usuario- Eliminación del usuario
Usuarios (<i>Registrados y verificados</i>) [uRV]	<ul style="list-style-type: none">- Mismo acceso que los <i>usuarios (No registrados y registrados)</i>- Creación de un blog- Publicación de artículos- Eliminación de artículos

Usuario Registrado

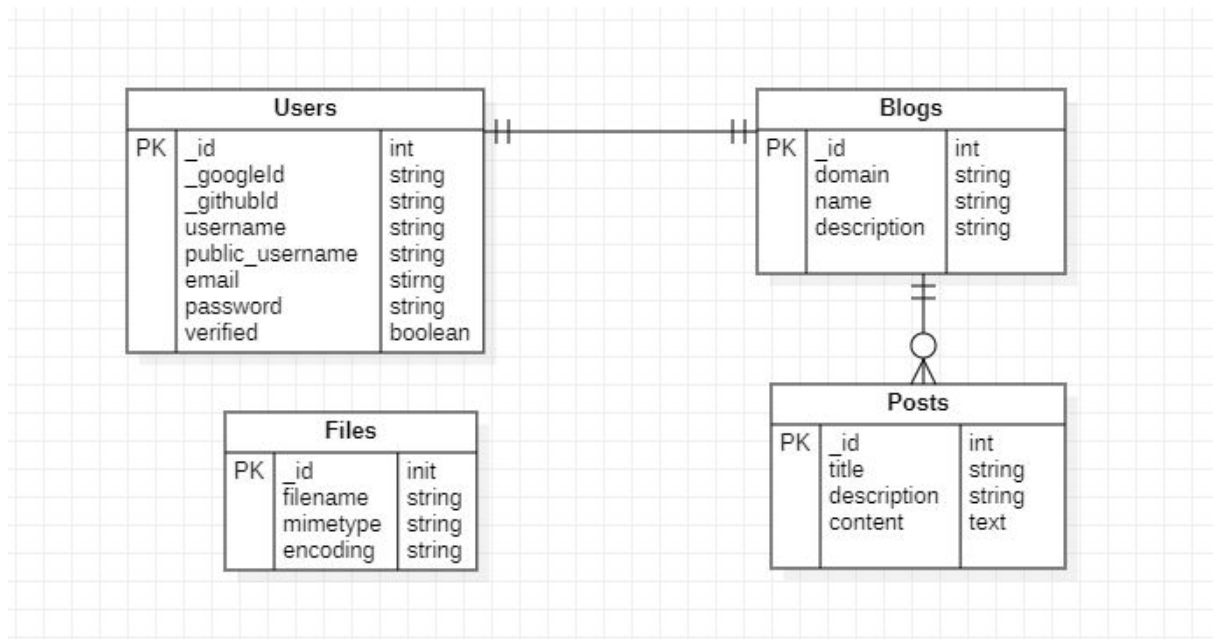


Entidad Relación

Definición del modelo de base de datos para implementar que el usuario pueda crear un blog y a su vez en este blog pueda crear varios artículos. Además para controlar los archivos subidos mediante GraphQL lo gestionaremos mediante otra entidad aparte.

Con la definición del diagrama de entidad relación podremos controlar los usos de coso mencionados anteriormente.

Diagrama de Entidad Relación



Control de versiones

Aparte para gestionar los archivos y un control de versiones usaremos Github donde almacenaremos en varios repositorios la gestión del proyecto tanto para back como para front.



Rutas

Rutas de acceso (back)

Rutas	Descripción	Acceso
/graphql	Acceso a peticiones al back	*
/auth/google	Petición passport google	*
/auth/google/callback	Google callback	*
/auth/github	Petición passport github	*
/auth/github/callback	GitHub callback	*

* - Dependiendo de la petición pedirá un token de usuario válido o sin necesidad de token.

Rutas de acceso (front)

Rutas	Descripción	Acceso
/	Página principal	*
/register	Página de registro de usuario.	uNO
/login	Página de autenticación de usuario.	uNO
/auth/oauth	Página para autenticar usuarios provenientes de peticiones OAuth	uNO
/logout	Página para desloguear un usuario	uR
/auth/verify/:userId	Página para verificar un usuario	uR
/account	Página con los datos del usuario y su posible modificación/descarga/eliminación de datos	uR
/new/blog	Página para la creación de un blog	uRV
/new/post	Página para la publicación de un artículo nuevo.	uRV
/blog/:blogDomain	Página de visualización de un blog	*
/blog/:blogDomain/:postId	Página de visualización de un artículo	*

* - Usuario registrado y usuario no registrado

uNO - Usuario no registrado

uR - Usuario registrado

uRV - Usuario registrado y verificado

Estructura de carpetas de GraphQL (back)

Carpetas

Auth: Archivos dedicados a la autenticación mediante OAuth.

Models: Modelos de mongoose para poder hacer peticiones a la base de datos.

Resolvers: Funciones que resolverán las queries pedidas.

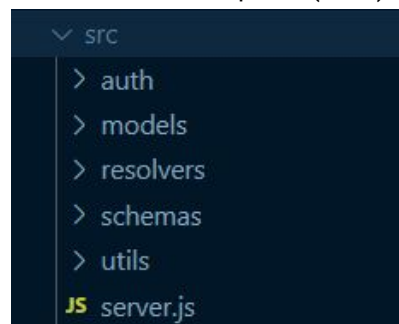
Schemas: Definición de modelos y queries permitidas de GraphQL.

Utils: Herramientas extra.

Archivos

server.js - Archivo principal que ejecuta el servidor de GraphQL.

Estructura de carpetas (back)



Estructura de carpetas de React (front)

Carpetas

Assets: Archivos multimedia o estilos css.

Components: Componentes reutilizables. (hooks)

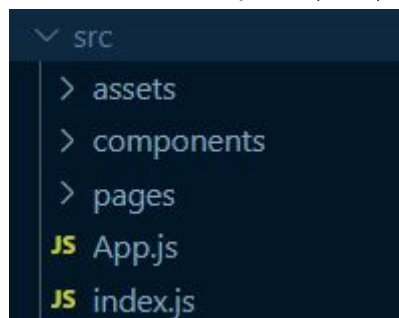
Pages: Componentes renderizan las páginas predefinidas. (hooks)

Archivos

App.js - Componente principal.

index.js - Archivo principal que renderiza al DOM.

Estructura de carpetas (front)



3.2.3 Implementación

Para comenzar un proyecto de GraphQL con la librería **Apollo** iniciaremos un proyecto con NodeJs

Comando init NodeJS

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-server
$ npm init -y
```

El comando te generará un archivo **package.json** donde ya podremos empezar instalando librerías.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-server
$ npm i apollo-server-express graphql express mongoose bcrypt dotenv
```

Antes de empezar configurando Apollo necesitaremos preparar el entorno para poder utilizar JavaScript moderno, para ello instalaremos y utilizaremos una librería llamada babel, además utilizaremos la librería nodemon para reiniciar automáticamente el servidor cada vez que hagamos un cambio en el código y así se apliquen los cambios.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-server
$ npm i -D @babel/cli @babel/core @babel/node @babel/preset-env nodemon
```

Después de instalar las dependencias de babel crearemos un archivo en la raíz del proyecto llamado **.babelrc** con la siguiente configuración

```
{
  "presets": [
    "@babel/preset-env"
  ]
}
```

Con babel y nodemon se encargan de compilar el código y ejecutarlo

```
"scripts": {
  "start": "nodemon ./src/server.js --exec babel-node"
},
```

Prepararemos un script de NodeJS para poder iniciar el servidor.

Código para iniciar servidor GraphQL [./src/server.js]

```
import { ApolloServer,
  AuthenticationError } from "apollo-server-express";
import express from "express";
import dotenv from "dotenv";

import schemas from "../schemas/schemas";
import resolvers from "../resolvers/resolvers";

dotenv.config();

const server = new ApolloServer({
  cors: { origin: '*' },
  playground: true,
  typeDefs: schemas,
  resolvers: resolvers,
  context: async ({ req }) => {
    const token = req.headers.authorization || '';
    let user = null;
    try {
      if(token) {
        user = jwt.verify(token.replace('Bearer ', ''),
          process.env.JWT_SECRET)
        return { token, user }
      }
    } catch(error) {
      throw new AuthenticationError(
        'Authentication token is invalid, please log in')
    }
    return {};
  }
});

const app = express();
server.applyMiddleware({ app });
app.listen({ port: process.env.APP_SERVER_PORT }, () => {
  console.log(`\n-> Server ready at
    ${process.env.APP_SERVER_URL}${server.graphqlPath}`);
});
```

En el archivo server.js es donde configuraremos el inicio del servidor GraphQL.

A la clase ApolloServer se le pasaran dos parámetros importantes typeDefs y resolvers, estos dos parámetros servirán para definir el esquema de las peticiones GraphQL y sus respectivos resolvers para realizar las peticiones a la base de datos.

```
typeDef de User [ ./src/schemas/user.schema.js]

import { gql } from "apollo-server-express";

export default gql`
  type User {
    _id: ID!
    _googleId: ID
    _githubId: ID
    username: String!
    public_username: String!
    email: String
    password: String
    avatar: String
    blog: Blog
    verified: Boolean
  }
// ...
```

Así definiríamos el tipo de datos de GraphQL, tiene que corresponder con el siguiente formato:

“nombre_del_dato”: tipo_de_dato

El único tipo de dato que puede resultar un poco confuso es **Blog** este tipo de datos se define de la misma manera que User.

```
typeDef de Blog [ ./src/schemas/blog.schema.js]

export default gql`
  type Blog {
    _id: ID!
    domain: String!
    name: String!
    description: String!
    author: User!
    posts: [Post!]!
  }
// ...
```

También en este tipado de datos se podría apreciar el signo de exclamación [!], se le pondrá a los datos que son obligatorios que los tenga en la base de datos para poder mostrar de forma correcta el dato pedido.

typeDef de User [./src/schemas/user.schema.js]

```
// ...
extend type Query {
  user: User!
  users: [User!]!
  sendVerificationEmail: Boolean
}

extend type Mutation {
  createUser(username: String!,
    email: String,
    password: String!): User!
  updateUser(public_username: String,
    email: String,
    password: String): User
  verifyUser(username: String,
    email: String,
    password: String): User
  deleteUser(_id: String): User
  register(email: String!,
    username: String!,
    password: String!): Token!
  login(email: String,
    password: String!): Token!
}
`;
```

También se le definirán las consultas y las mutaciones para poder modificar los datos y poder consultarlos.

Antes de definir nuestro resolver es necesario definir el modelo de MongoDB con la librería mongoose y así poder realizar peticiones a la base de datos.

Conectar mongoose con MongoDB [./src/server.js]

```
import mongoose from "mongoose";

mongoose.connect(`mongodb://${process.env.DATABASE_URL}:${process.env.DATABASE_PORT}/${process.env.DATABASE_NAME}`, {
  useUnifiedTopology: true,
  useNewUrlParser: true,
  useCreateIndex: true
});
```

Definición de un modelo con mongoose [./src/models/user.js]

```
import mongoose from "mongoose";
import bcrypt from "bcrypt";

const userSchema = new mongoose.Schema({
  _googleId: { type: Number, unique: false },
  _githubId: { type: String, unique: false },
  username: { type: String, required: true, unique: true },
  public_username: { type: String, unique: true },
  email: { type: String },
  password: { type: String, required: true },
  verified: { type: Boolean, default: false }
});

userSchema.pre('save', function() {
  this.password = bcrypt.hashSync(this.password, 12);
  this.public_username = this.username;
});

export default mongoose.model("user", userSchema);
```

Configuraremos mongoose para que se pueda conectar con MongoDB y le definiremos los modelos de la siguiente manera. Un tipo de datos bastante sencillo, donde si le defines si es único o requerido se encargará de generar un Index en MongoDB.

Cabe recalcar la función *pre* que se ejecutará cada vez que se guarde un registro, así podremos encriptar las contraseñas.

Definición de queries de los typeDefs [./src/resolvers/user.resolver..js]

```
import User from "../models/user";

export default {
  Query: {
    users: async (root, args, { user }) => {
      if(!user) {
        throw new AuthenticationError("You are not logged in"); }
      return await User.find().exec();
    },

    user: async (root, args, { user }) => {
      if(!user) {
        throw new AuthenticationError("You are not logged in"); }
      return await User.findById(user.user._id).exec();
    },
  },
},
```

En el resolver de se encargará de llamar la base de datos y pedir los datos además de poder hacer comprobaciones antes de la petición.

Definición de mutations de los typeDefs [./src/resolvers/user.resolver..js]

```
Mutation: {
  deleteUser: async (root, args, { context }) => {
    if(!user) {
      throw new AuthenticationError("You are not logged in"); }
    return await User.findByIdAndRemove(context.user._id);
  },
},
```

Las mutaciones también se definen de la misma manera.

Definición de tipo de dato de los typeDefs [./src/resolvers/user.resolver..js]

```
User: {
  blog: async (root, args) => {
    return await Blog.findOne({ author: root._id }).exec();
  }
}
```

De esta manera podemos llamar al tipo de dato mencionado anteriormente y así poder hacer una única petición.

Desde este punto tendríamos definido GraphQL pero para complementar los requerimientos es necesario la configuración de email y autenticación de usuario mediante OAuth.

Instalacion de librerias

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-server
$ npm i nodemailer passport passport-google-oauth passport-github2
```

Configuración email [./src/utils/mail.js]

```
import nodemailer from "nodemailer";
import hbs from "nodemailer-express-handlebars";

const configMail = (type) => {
  let transporter = nodemailer.createTransport({
    host: 'smtp.gmail.com',
    port: 465,
    secure: true,
    auth: {
      type: 'OAuth2',
      user: process.env.EMAIL,
      clientId: process.env.GOOGLE_CLIENT_ID,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET,
      refreshToken: process.env.GOOGLE_REFRESH_TOKEN,
      accessToken: process.env.GOOGLE_ACCESS_TOKEN,
    },
  });

  transporter.use('compile', hbs({
    viewEngine: {
      extName: '.handlebars',
      partialsDir: `./src/utils/emails/${type}` || './src/utils/emails',
      layoutsDir: `./src/utils/emails/${type}` || './src/utils/emails',
      defaultLayout: 'index.handlebars'
    },
    viewPath: `./src/utils/emails/${type}` || './src/utils/emails',
    extName: '.handlebars',
  }));

  return transporter;
}
```

Con la librería **nodemailer** de esta manera nos conectaremos con el servicio de SMTP de google para poder realizar el envío de emails y con la librería **hbs** podremos usar plantillas html y hacer mails personalizados y haremos una función simple para el envío de mails.

Función sendEmail [./src/utils/mail.js]

```
const sendEmail = (user, subject, type) => {
  let transporter = configMail(type);
  let mailOption = {
    from: process.env.EMAIL,
    to: user.email,
    subject: subject,
    template: 'index',
    context: {
      userId: user._id,
      name: user.username
    }
  }
  transporter.sendMail(mailOption);
}
```

Ya con todo preparado y configurado ya podemos enviar emails de manera sencilla simplemente usando la función.

Ejemplo de uso de la función de mail

```
sendEmail(context.user, `Bienvenido ${user.user.username}!`, 'verify');
```

Para poder iniciar una autenticación OAuth utilizaremos **Passport** necesitaremos inicializarlo y configurarlo.

Passport initialize [./src/auth/index.js]

```
import passport from 'passport';

passport.serializeUser(function(user, done) { done(null, user._id); });
passport.deserializeUser(function(user, done) { done(null, user); });

app.use(passport.initialize());
app.use(passport.session());
```


Para la configuración del OAuth de google necesitaremos las credenciales que nos proporcionan en [Google APIs](<http://console.developers.google.com>), una vez nos devuelva los datos pedidos nos guardaremos en nuestra base de datos como usuario.

Passport google config [./src/auth/google/index.js]

```
import { OAuth2Strategy as GoogleStrategy } from
"passport-google-oauth";

passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: `/auth/google/callback`
}, async function(accessToken, refreshToken, profile, done) {
  const user = User.findOneOrCreate({ _googleId: profile.id }, {
    _googleId: profile.id,
    username: profile.name.givenName.toLowerCase(),
    email: profile.emails[0].value,
    password: profile.id
  });
  user.then(user => {
    return done(null, user);
  });
}));
```

Necesitaremos dos direcciones url para que google nos envíe los datos, la primera url `/auth/google` se utilizara para que el usuario pueda mandar una petición OAuth a google.

Ruta petición OAuth google [./src/auth/google/index.js]

```
app.get('/auth/google',
  passport.authenticate('google',
    { scope: ['https://www.googleapis.com/auth/plus.login',
      'https://www.googleapis.com/auth/userinfo.email'] }));
```

También necesitaremos una url `/auth/google/callback` para que cuando google nos mande los datos poder realizar acciones al cliente.

Generaremos un token JWT y se lo pasaremos al cliente mediante las cookies.

Ruta callback OAuth google [./src/auth/google/index.js]

```
import User from "../../models/user";
import jwt from "jsonwebtoken";

app.get('/auth/google/callback',
  passport.authenticate('google'), function(req, res) {
    const user = req.user;
    const token = jwt.sign(
      {
        user: {
          _id: user._id,
          email: user.email,
          username: user.username,
          public_username: user.public_username,
        },
      },
      process.env.JWT_SECRET,
      { expiresIn: "1d" }
    );

    res.status(200).cookie('OAuthUser',
      JSON.stringify(user), { maxAge: (1 * 3600 * 1000) });
    res.status(200).cookie('OAuthToken',
      token, { maxAge: (24 * 3600 * 1000) });
    res.redirect(`${process.env.APP_CLIENT_URL}/auth/oauth`);
  });
```

Con esto tendremos listo la configuración de las librerías en el back para poder pedir datos desde front con peticiones GraphQL

A continuación para poder realizar un proyecto de React utilizaremos una librería llamada **create-react-app** ya que como mencione anteriormente te otorga una configuración para el desarrollo con React.

Comando init create-react-app

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npx create-react-app .
```

Nos generara un proyecto con varios archivos, los cuales nos quedaremos con los necesarios.

Directorio de front



Nos generara un proyecto con varios archivos, los cuales nos quedaremos con los archivos necesarios para poder realizar el proyecto.

Configuraremos React con la librería de **Apollo Client** para poder realizar peticiones de manera sencilla.

función para enviar el token [./src/index.js]

```
const request = async (operation) => {
  const token = localStorage.getItem('token');
  operation.setContext({
    headers: {
      authorization: token ? `Bearer ${token}` : '',
    }
  });
};
```

Prepararemos una función para que en cada petición envíe el token del usuario.

Instalaremos todas las dependencias necesarias para la configuración y uso de Apollo Client y así poder realizar peticiones GraphQL.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npm i apollo-client apollo-link apollo-cache-inmemory
apollo-upload-client @apollo/react-hooks react-apollo graphql-tag
```

Definimos ApolloLink para poder hacer observer de todas las peticiones que hagamos y se quede escuchando la petición e inicializamos **ApolloClient**.

Configuración de ApolloClient con ApolloLink [./src/index.js]

```
import ApolloClient from 'apollo-client';
import { ApolloLink, Observable } from 'apollo-link';
import { createUploadLink } from 'apollo-upload-client';
import { InMemoryCache } from 'apollo-cache-inmemory';

const requestLink = new ApolloLink((operation, forward) =>
  new Observable(observer => {
    let handle;
    Promise.resolve(operation)
      .then(oper => request(oper))
      .then(() => {
        handle = forward(operation).subscribe({
          next: observer.next.bind(observer),
          error: observer.error.bind(observer),
          complete: observer.complete.bind(observer),
        });
      }).catch(observer.error.bind(observer));
    return () => { if (handle) handle.unsubscribe(); };
  }));

const client = new ApolloClient({
  link: ApolloLink.from([
    requestLink,
    new createUploadLink({
      uri: `${process.env.REACT_APP_API_URL}/graphql`,
    })
  ]),
  cache: new InMemoryCache()
});
```

Con `createUploadLink` podremos enviar al back una imagen a modo de petición GraphQL y una vez configurado `ApolloClient` se lo pasaremos al provider para renderizarlo.

Renderizar la aplicación en elemento `#root` [./src/index.js]

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
  <ApolloProvider client={ client }>
    <App />
  </ApolloProvider>,
  document.getElementById('root')
);
```

Antes de empezar con la lógica de la aplicación configuraremos **Tailwind Css** con PostCSS para darle estilo al proyecto.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npm i -D @fullhuman/postcss-purgecss postcss-cli postcss-import
autoprefixer concurrently tailwindcss @tailwindcss/custom-forms
```

Crearemos un archivo donde importaremos de tailwind los estilos css este archivo servirá para que postcss pueda compilar el archivo css.

Importacion de tailwindcss [./src/assets/tailwind.css]

```
@import "tailwindcss/base";
@import "tailwindcss/components";
@import "tailwindcss/utilities";
```

Como comentamos anteriormente tailwind crea demasiadas clases ya que es una librería donde estilizas el css mediante clases, tailwind por sí solo tiene +65.000 líneas de código css, es por eso que configuraremos con postcss la librería **purgecss** lo que hará es ver todas las clases utilizadas y cuando se haga un build del proyecto eliminará las clases no utilizadas en el proyecto, además configuraremos autoprefixer para añadir estilos generales para todos los navegadores.

Primero inicializamos tailwind que nos generará un archivo llamado tailwind.config.js en la raíz del proyecto que nos permitirá configurar los estilos de tailwindcss.

Inicialización de tailwind

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npx tailwindcss init --full
```

Y luego crearemos un archivo llamado postcss.config.js en la raíz del proyecto para la configuración de PostCSS comentada anteriormente.

Configuración de PostCSS [./postcss.config.js]

```
let purgecss = require('@fullhuman/postcss-purgecss');

module.exports = {
  plugins: [
    require('postcss-import'),
    require('tailwindcss'),
    require('autoprefixer'),
    ...process.env.NODE_ENV === 'production' ? [
      purgecss({
        content: [
          './public/index.html',
          './src/**/*.js',
        ],
        css: ['./src/assets/css/main.css'],
        defaultExtractor: content => content.match(/[\w-/:]+(?
```

Crearemos dos scripts de NodeJS para poder generar el archivo de estilo de tailwind

```
"scripts": {
  "start:tailwind": "postcss ./src/assets/css/tailwind.css -o ./src/assets/css/main.css -w",
  "build:tailwind": "postcss ./src/assets/css/tailwind.css -o ./src/assets/css/main.css --env production",
},
```

Vamos a empezar configurando las rutas del proyecto definidas en Diseño para ello necesitaremos una librería llamada **react-router-dom** además aprovecharemos e instalaremos una librería para controlar los formularios para más adelante.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npm i react-router-dom react-hook-form
```

Rutas definidas con componentes hooks [./src/App.js]

```
function App() {
  return (
    <Router>
      <Header/>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/register" component={Register} />
        <Route path="/login" component={Login} />
        <Route path="/auth/oauth" component={OAuth} />
        <Route path="/auth/verify/:userId" component={VerificationEmail} />
        <Route path="/new/blog" component={NewBlog} />
        <Route path="/new/post" component={NewPost} />
        <Route path="/blog/:blogDomain/:postId" component={Post} />
        <Route path="/blog/:blogDomain" component={Blog} />
        <Route component={Error} />
      </Switch>
      <Footer />
    </Router>
  );
}

export default App;
```

Para la definición de un componente en React utilizando hooks se realizaría de la siguiente manera, importamos React y exportamos por defecto una función que retorna html definido en JSX.

(JSX attribute) `React.HTMLAttributes<HTMLDivElement>.className?: string`

Definiendo componente hook en React [./components/Error.js]

```
import React from 'react';

function Error() {
  return (
    <div className="px-4">
      <div className="max-w-3xl mx-auto my-16 p-16">
        <h1 className="text-6xl font-medium mb-2 text-center">Error 404</h1>
        <p className="text-2xl text-center">Página no encontrada</p>
      </div>
    </div>
  );
}

export default Error;
```

Ya que el proyecto tiene bastante lógica no podré explicarlo todo en exactitud pasará a explicar funciones esenciales en el proyecto, como gestión de formularios, editor de html para generar los artículos y peticiones GraphQL desde el lado del cliente con la librería Apollo Client.

Definiendo esquema para una petición en GraphQL [./page/Auth/Register.js]

```
import gql from 'graphql-tag';

const REGISTER = gql`
  mutation register($username: String!,
    $email: String!, $password: String!) {
    register(
      username: $username
      email: $email
      password: $password
    ) { token }
  }
`;
```


Definiendo useMutation [./page/Auth/Register.js]

```
import { useMutation } from '@apollo/react-hooks';

function Register(props) {
  const [ registerUser, { error, data } ] = useMutation(REGISTER,
    { onCompleted: onCompleted });
  // ...
}
```

Para realizar una mutación lo primero que tenemos que realizar es la definición de la query con un namespace y para la utilización de esa query utilizaremos useMutation donde te otorga una función para realizar la mutación y variables para acceder a los datos o el mensaje de error.

Uso de useForm [./page/Auth/Register.js]

```
import { useForm } from 'react-hook-form';

const onSubmit = (data) => {
  registerUser({ variables: { username: data.username,
    email: data.email, password: data.password } });
}

const { register, handleSubmit, errors } = useForm();
return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <label className="block max-w-3xl mx-auto px-16 py-2 mt-8">
      <span className="text-gray-700 ml-1">Username:</span>
      <input type="text" name="username" ref={register({
        required: true })} className="form-input mt-1 block w-full" ></input>
      { errors.username && <span>This field is required</span> }
    </label>
    // ...
  </form>
)
```

La utilización de useForm sería de la siguiente manera, crearemos una función que se ejecuta después de hacer submit se lo pasaremos a la etiqueta form como parámetro, además los valores register y errors son para registrar un campo y aplicarles requerimientos si no cumplen con esos requerimientos el formulario no se enviará además podremos asignar mensajes personalizados para cada campo con la variable errors..

Para realizar una query es similar a una mutación, lo primero que habría realizar es la definición de la query.

Uso de useForm [./page/Blog/Blog.js]

```
import gql from 'graphql-tag';

const GET_BLOG = gql`  query blog($domain: String!) {
  blog(domain: $domain) {
    _id
    domain
    name
    description
    author {
      _id
      username
    }
    posts {
      _id
      title
      description
    }
  }
}`;
```

Pero a diferencia de la mutación la query se ejecuta una vez se define las variables de useQuery y se queda escuchando, por ello habría que controlar la vista hasta que traiga los datos.

Uso de useForm [./page/Blog/Blog.js]

```
import { useQuery } from '@apollo/react-hooks';
import { useParams } from 'react-router-dom';

function Blog(props) {
  let { blogDomain } = useParams();
  const { data, loading, error } = useQuery(GET_BLOG, { variables: {
    domain: blogDomain } })
  if(loading) return null;
  if(error) return <Error />;
  // ...
}
```

Para poder realizar este proyecto en front es necesario tener un editor de texto enriquecido para ello usaremos la librería **EditorJS** y algunos plugins que tiene para poder realizar la edición de un artículo.

Instalación de librerías

```
Dani@DESKTOP-V4QJ9BA MINGW64 /c/project/dblogs-client
$ npm i -D @editorjs/editorjs @editorjs/delimiter @editorjs/embed
@editorjs/header @editorjs/image @editorjs/link @editorjs/list
@editorjs/paragraph @editorjs/quote @editorjs/table @editorjs/marker
@editorjs/warning
```

Prepararemos un archivo de configuración para poder utilizar todos los plugins y le definiremos un nombre el cual utilizara para generar un json y luego nosotros podremos renderizar ese json.

Configuración EditorJS [./src/components/EditorJS/config/tools.js]

```
import Embed from '@editorjs/embed'
import Table from '@editorjs/table'
import List from '@editorjs/list'
import Warning from '@editorjs/warning'
import LinkTool from '@editorjs/link'
import Header from '@editorjs/header'
import Marker from '@editorjs/marker'
import Quote from '@editorjs/quote'
import CodeTool from '@editorjs/code'
import Delimiter from '@editorjs/delimiter'
import Image from '@editorjs/image'
export const EDITOR_JS_TOOLS = {
  embed: { class: Embed, },
  header: { class: Header, inlineToolbar: true, },
  list: { class: List, },
  table: { class: Table, },
  warning: { class: Warning, },
  linkTool: { class: LinkTool, },
  marker: { class: Marker, },
  delimiter: { class: Delimiter, },
  image: { class: Image, },
  quote: { class: Quote, },
  code: { class: CodeTool, },
}
```

Para poder subir una imagen en EditorJS mediante GraphQL prepararemos una mutación en el componente y añadiremos el método a la configuración de EditorJS.

Esquema para la subida de archivos [./src/pages/Blog/NewPost.js]

```
const UPLOAD_FILE = gql`mutation($file: Upload!) {
  uploadFile(file: $file) {
    _id
    filename
  }
}`;
```

Configuración EditorJS para la subida de imágenes [./src/pages/Blog/NewPost.js]

```
EDITOR_JS_TOOLS.image = {
  class: Image,
  config: {
    uploader: {
      uploadByFile: (file) => {
        return uploadFile({ variables: { file: file } }).then(image => {
          return {
            "success" : 1,
            "file": {
              "url": process.env.REACT_APP_API_URL + "/images/" +
                image.data.uploadFile._id + "-" +
                image.data.uploadFile.filename,
            }
          }
        })
      }
    }
  }
}
```

La utilización del editor es bastante sencilla, simplemente agregamos el elemento en nuestro return de JSX

Utilización del componente EditorJS [./src/pages/Blog/NewPost.js]

```
import EditorJs from 'react-editor-js';

function NewPost(props) {
  //...
  return (
    //...
    <EditorJs tools={ EDITOR_JS_TOOLS } placeholder="Escribe tu hi..." />
    //...
  )
}
```

EditorJS al guardarlo nos generará un JSON del siguiente formato.

Ejemplo JSON generado por EditorJS

```
{
  "time" : 1590298784949,
  "blocks" : [
    { "type" : "header",
      "data" : {
        "text" : "Editor.js",
        "level" : 2
      }
    },
    { "type" : "paragraph",
      "data" : {
        "text" : "Hey. Meet the new Editor. On this page yo..."
      }
    }
  ]
}
```

Con esto en mente prepararemos un componente para poder renderizar este JSON a la vista.

Ejemplo de renderizado del header [./src/components/EditorJSRender.js]

```
import React from 'react';

function EditorJsRender(props) {
  return (
    <>
      { props.render.blocks.map((block, i) => {
        return <div className="ce-block" key={ i }>
          <div className="ce-block__content">
            { (block.type === "header") &&
              React.createElement(`h${block.data.level}`, {
                className: "ce-header" }, block.data.text)
            }
          </div>
        }
      )
    </>
  );
}
```

Utilización del componente EditorJSRender [./src/pages/Blog/Post.js]

```
import { EditorJsRender } from '../../components';

function NewPost(props) {
  //...
  return (
    //...
    <EditorJsRender render={JSON.parse(data.post.content)} />
    //...
  );
}
```

Faltaría explicar bastante lógica del proyecto en front pero con todo la explicación otorgada sería replicar y utilizar todo lo explicado anteriormente.

Está disponible el código en github para visualizar toda la lógica del proyecto.

[d Blogs Server en Github](<https://github.com/dmarine/dblogs-server>)

[d Blogs Client en Github](<https://github.com/dmarine/dblogs-client>)

3.2.3 Pruebas

Las pruebas de funcionamiento del proyecto han sido a base de prueba y error, por el simple hecho de falta de tiempo pero me hubiera gustado implementar los testing de **Jest** y la librería de **Apollo Client Testing** para testear la aplicación de React en front y para GraphQL en back la librería de **Apollo Server Testing**.

[Jest](<https://jestjs.io/docs/es-ES/tutorial-react>)

[Apollo](<https://www.apollographql.com/docs/react/development-testing/testing/>)

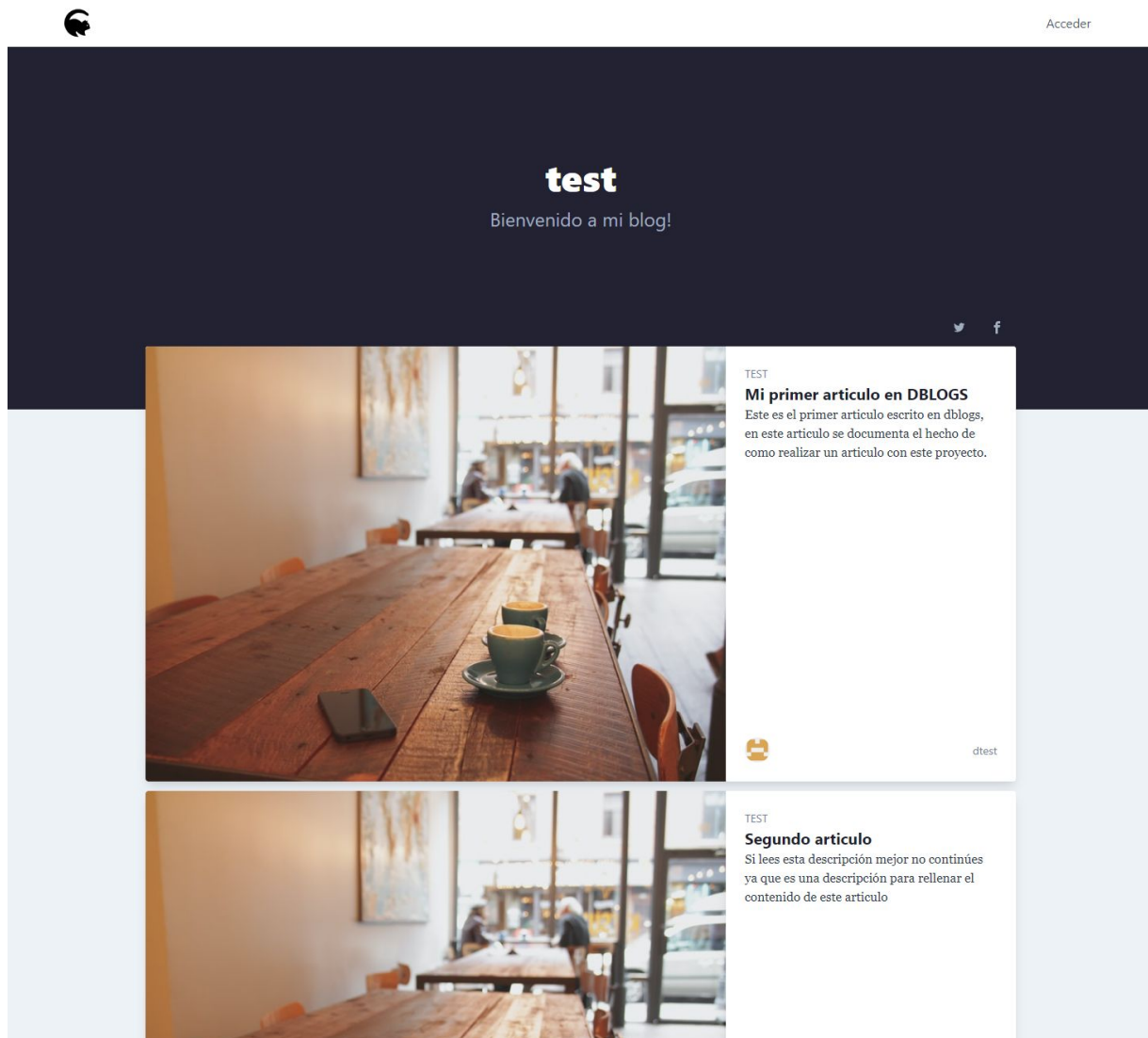
[Apollo

Testing](<https://www.apollographql.com/docs/apollo-server/testing/testing/>)

4. Manual de uso

El uso básico, navegación por un blog.

Vista de un blog en dblogs



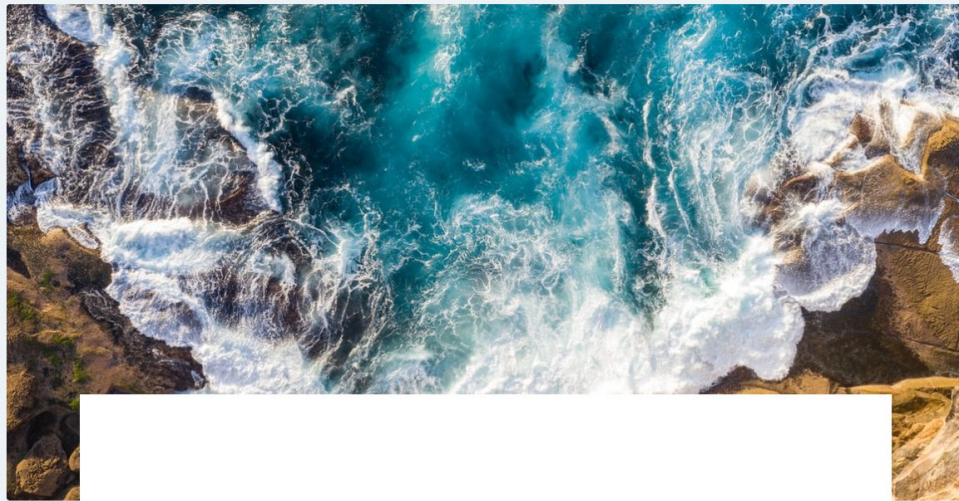
Desde un vistazo puedes observar el título del blog y sus artículos simplemente entra a un artículo y entretente con su contenido.

También podrías compartir el blog mediante los iconos de las redes sociales.



08 ABRIL 2020 / TEST

Mi primer articulo en DBLOGS



Hola! Bienvenido a mi blog



Después de haber leído el artículo al completo volver al blog es tan sencillo como darle a la flecha ubicada arriba a la izquierda y busca más artículos interesantes del mismo blog.



Login

Email:

Password:

[Login](#)[¿Aún no tienes cuenta? Regístrate aquí](#)[Sign up with Google](#)[Sign up with Github](#)

Regístrate

Username:

Email:

Password:

[Crear cuenta](#)[¿Ya tienes cuenta? Logeate aquí](#)


La autenticación en el sitio es bastante sencilla puedes registrar tu mismo en el link ubicado en “¿Aún no tienes cuenta? Regístrate aquí” o loguearse mediante autenticación OAuth (google, github)

 Iniciar sesión con Google

Selecciona una cuenta


para ir a [dblogs](#)


 **Daniel Marín Egea**
danielmarinegea@gmail.com

 Usar otra cuenta

Para continuar, Google compartirá tu nombre, tu dirección de correo electrónico, tu preferencia de idioma y tu foto de perfil con dblogs.

Español (España) ▾ Ayuda Privacidad Términos





Sign in to GitHub

to continue to [dblogs](#)

Username or email address

Password [Forgot password?](#)

[Sign in](#)

New to GitHub? [Create an account.](#)

Si lo geas mediante la autenticación OAuth y no tiene todos los datos necesarios para que un usuario funcione de forma correcta te pedirá todos los datos necesarios y una contraseña para poder acceder.

Autenticación OAuth

Crear Blog 

[Cerrar Sesión](#)

Datos necesarios

- Estos datos son necesarios para proceder con el registro -

Email:

Contraseña:

Actualizar

Si te identificas mediante OAuth tu cuenta será verificada automáticamente pero si te registras mediante un registro normal será necesario verificar la cuenta para crear un blog, para ello cuando te registres será enviado un mail.

Crear Blog 

[Cerrar Sesión](#)

Nuevo Blog

Volyster

Volyster@gmail.com

Dominio

Título

Descripción

Enviar

Necesitas verificar tu cuenta para poder crear un blog

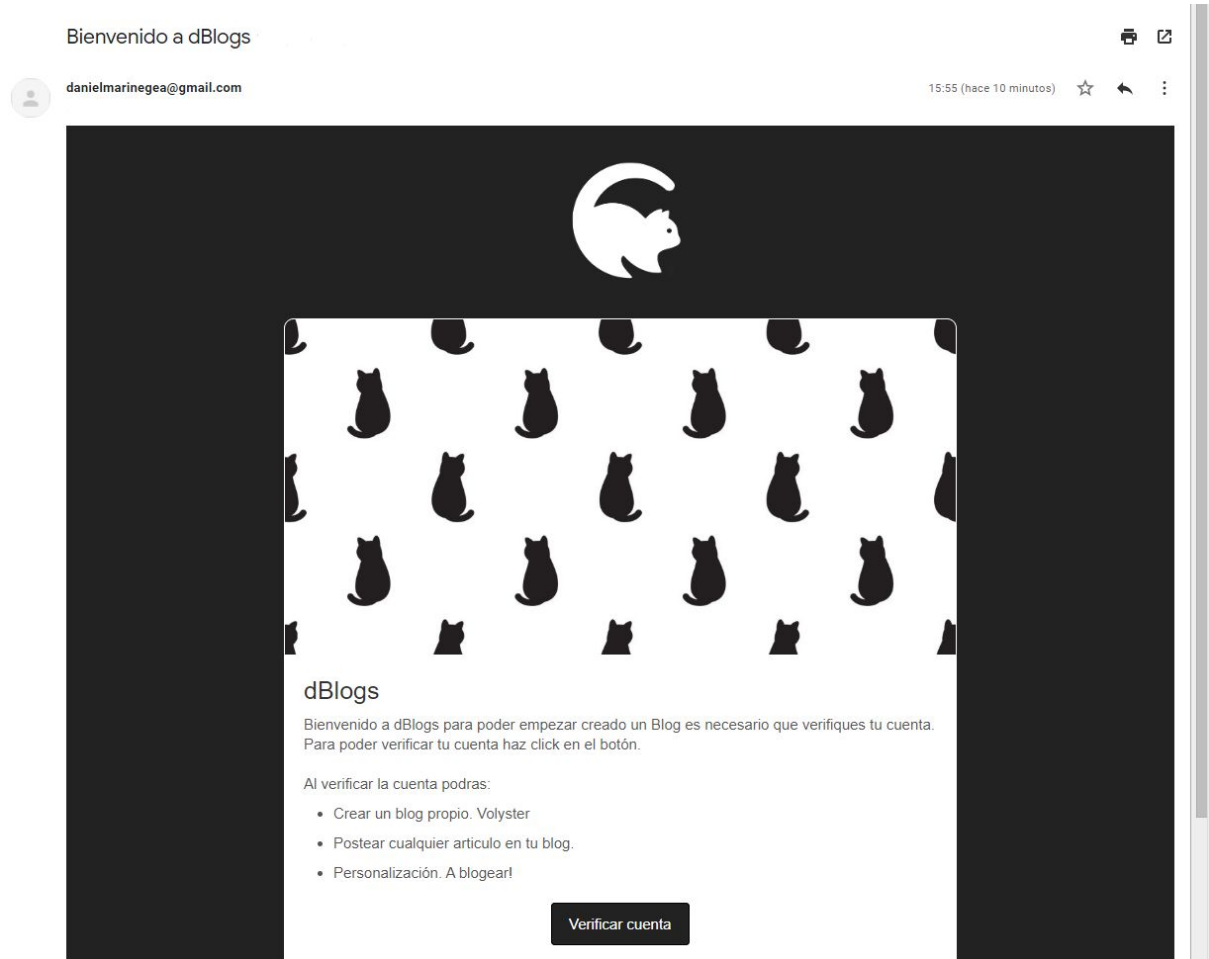
¿No has recibido el correo electrónico? [Haz click aquí para volver a enviar el correo de verificación](#)

  **danielmarinegea**

Bienvenido a dBlog Volyster! - Bienvenido a dblogs! Verifica tu cuenta para poder tener acces...

15:55

Email de verificación



En el mail junto con la información tendremos un botón de verificación que nos redirigirá a la web para proceder con la verificación.

Página de verificación



En el mail junto con la información tendremos un botón de verificación que nos redirigirá a la web para proceder con la verificación.

Una vez verificada la cuenta podemos proceder a la creación de un blog.

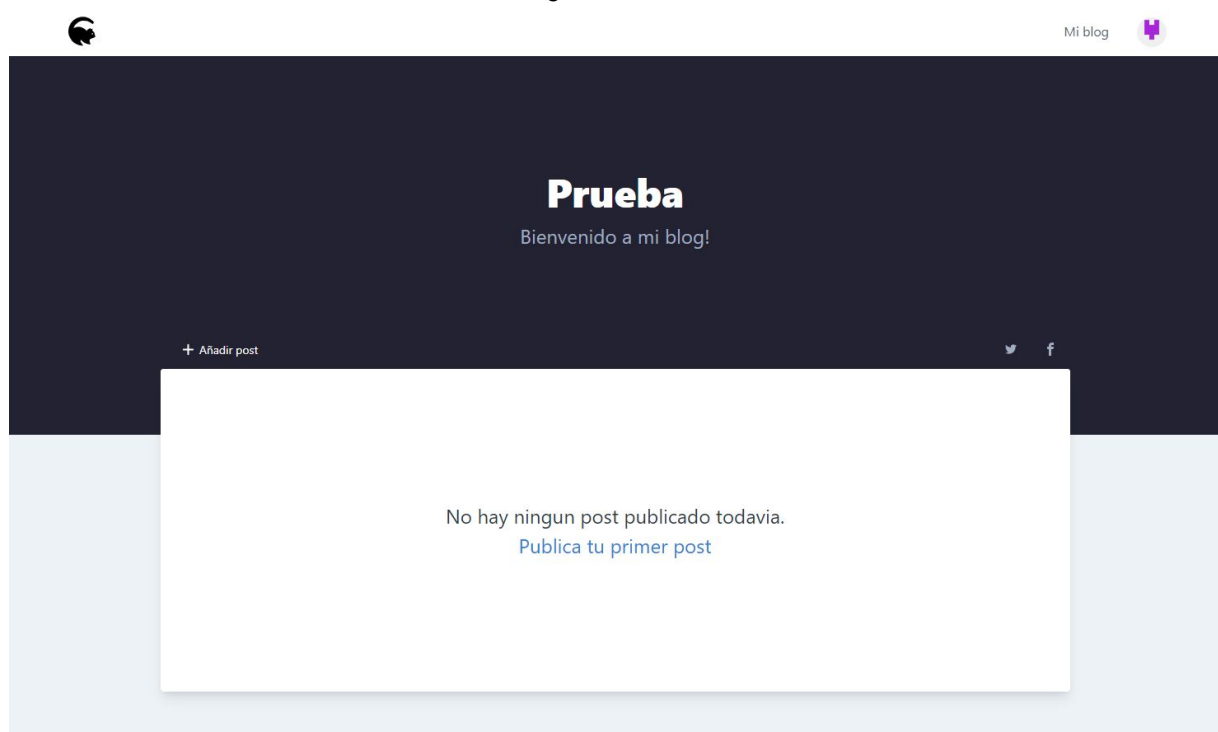
Simplemente una vez logeado en el menú de navegación podrás acceder rápidamente a la creación de un blog

Página de creación de un blog



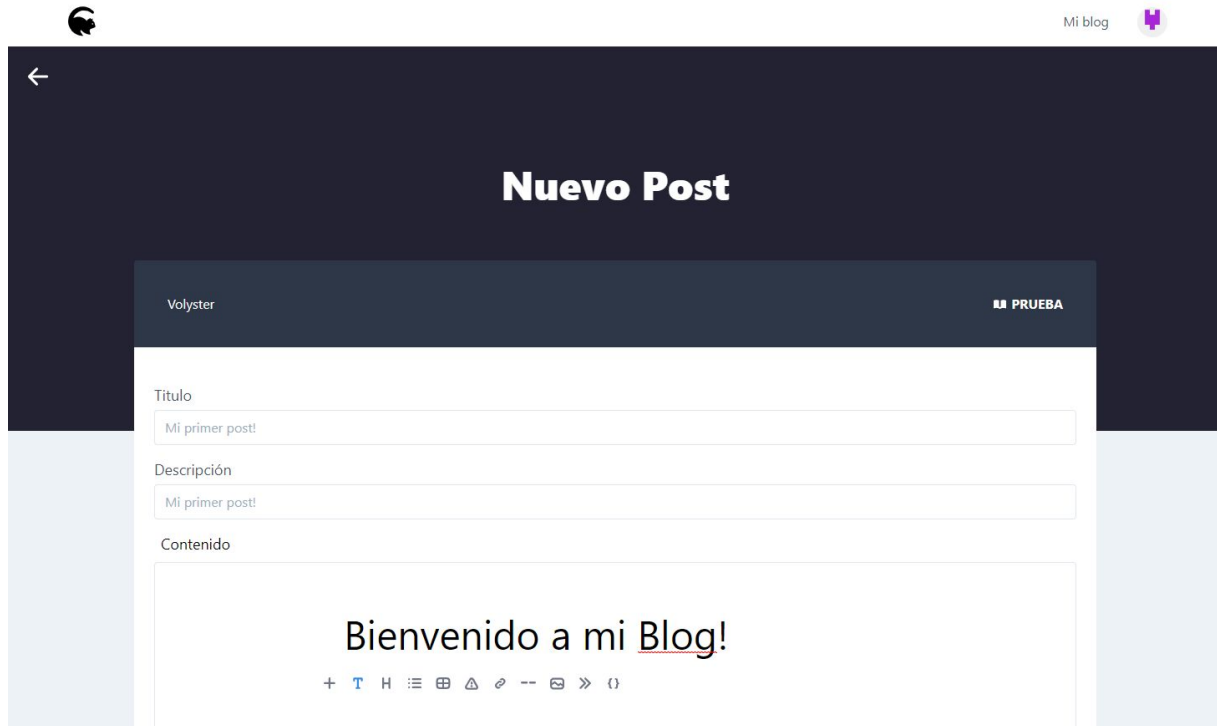
Simplemente una vez logeado en el menú de navegación podrás acceder rápidamente a la creación de un blog, donde te pedira el dominio de tu blog, título y la descripción.

Blog sin artículos

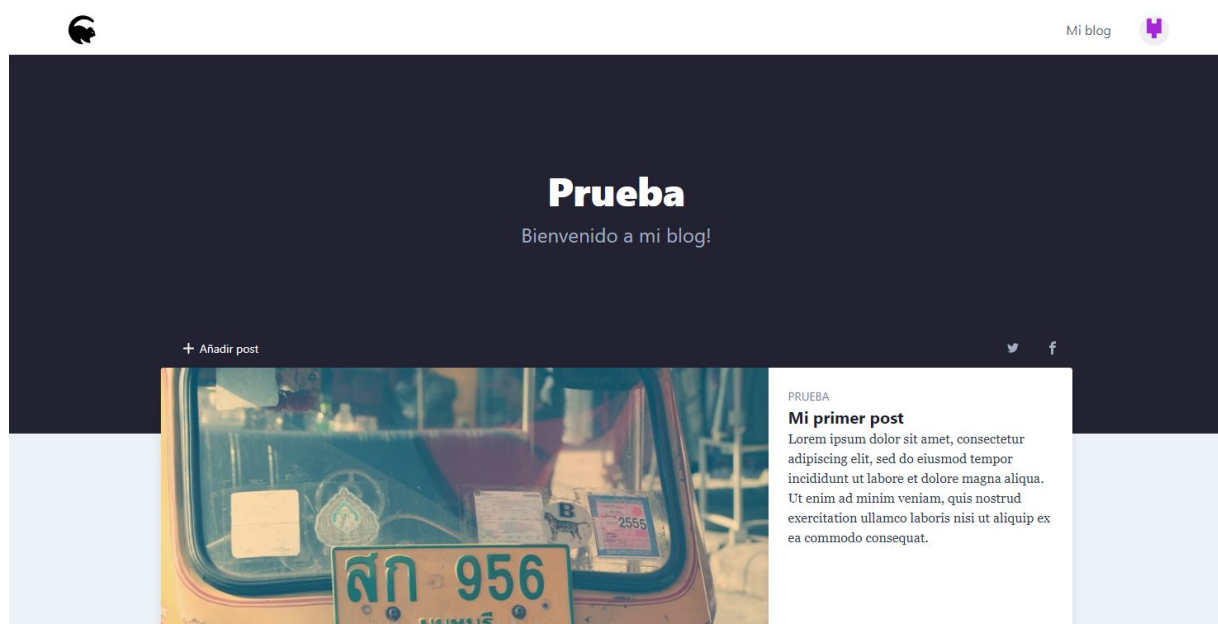


Una vez creado el blog podremos public artículos desde el botón “Añadir post”, donde podremos escribir nuestro artículo con un textarea con texto enriquecido.

Pagina para crear un nuevo artículo



Artículo publicado



5. Valoración personal del proyecto

Documentando el proyecto me he dado cuenta de lo que realizando una investigación se aprende de programación no tan solo picando código sino de su lógica y conceptos nuevos.

Realizar el proyecto basado en GraphQL y tener que enviar un json en vez de hacer varias urls para hacer peticiones es un concepto que personalmente me gustó bastante, lo único que para realizar una autenticación basada en OAuth era necesario crear urls para indicarle a los sitios de terceros donde tenían que realizar la petición, me hubiera gustado haber tenido la posibilidad de hacer la petición OAuth basada en GraphQL.

En cuanto a rendimiento y funcionalidades en el back estoy satisfecho y puede realizar todo lo que me propuse, como envío de mails, autenticación basada en OAuth o peticiones a una base de datos no relacional etc..., pero en front me hubiera gustado tener un poco más de tiempo para poder realizar todo lo que me hubiera gustado hacer.

Mi propuesta de mejora dedicada en lo dicho anteriormente en cuanto a funcionalidad es poder realizar tests basados en Jest y poder implementar la función de multi idioma basado en i18next, pero también me hubiera gustado poder refactorizar el código ya que cuando empiezas un proyecto sin ningún tipo de conocimiento luego vas aprendiendo y encontrando mejores maneras de hacer distintos tipos de la lógica.

Para los siguientes cursos me gustaría que propongáis la idea de una buena organización del proyecto, por ejemplo la utilización de programas externos como es Trello, escribirse todas las funcionalidades que tienen que hacer y luego ver que todas las tarjetas están en el apartado DONE e indicando las horas utilizadas para cada tarjeta y ver lo que mas te a costado realizar.

También agradecer a todas las personas que me rodean ya que opino que han aportado de varias maneras el poder realizar este proyecto, ya sea enseñándome programación, lanzando retos de programación o simplemente estando ahí.

6. Fuentes Bibliográficas

Fuentes bibliográficas con el estándar Mastering Markdown

[Javascript MDN](<https://developer.mozilla.org/es/docs/Web/JavaScript>)
[NodeJS](<https://nodejs.org/es/>)
[NPM](<https://www.npmjs.com/>)
[Git](<https://git-scm.com/>)
[GitHub](<https://github.com/>)
[Express](<https://expressjs.com/es/>)
[GraphQL](<https://graphql.org/>)
[React JS](<https://es.reactjs.org/>)
[Create React App](<https://create-react-app.dev/>)
[Webpack](<https://webpack.js.org/>)
[Apollo GraphQL](<https://www.apollographql.com/>)
[Apollo Server](<https://www.apollographql.com/docs/apollo-server/>)
[Apollo Client](<https://www.apollographql.com/docs/react/>)
[Tailwind CSS](<https://tailwindcss.com/>)
[Post CSS](<https://postcss.org/>)
[Purge CSS](<https://purgecss.com/>)
[Autoprefixer CSS](<https://autoprefixer.github.io/>)
[EditorJS](<https://editorjs.io/>)
[Stack Overflow](<https://stackoverflow.com/>)
[Medium](<https://medium.com/>)
[JWT](<https://jwt.io/>)
[Mongo DB](<https://www.mongodb.com/es>)
[Mongoose JS](<https://mongoosejs.com/>)
[Passport JS](<http://www.passportjs.org/>)
[Babel JS](<https://babeljs.io/>)
[Nodemailer](<https://nodemailer.com/about/>)