**Universidad San Carlos de Guatemala**
**Facultad de Ingeniería**
**Escuela de Ciencias y Sistemas**
**Análisis y Diseño1, Sección N**
**Catedrático:** Ing. Sergio Mendez
**Aux: Leonel Aguilar**

# MANUAL TECNICO PROYECTO 1



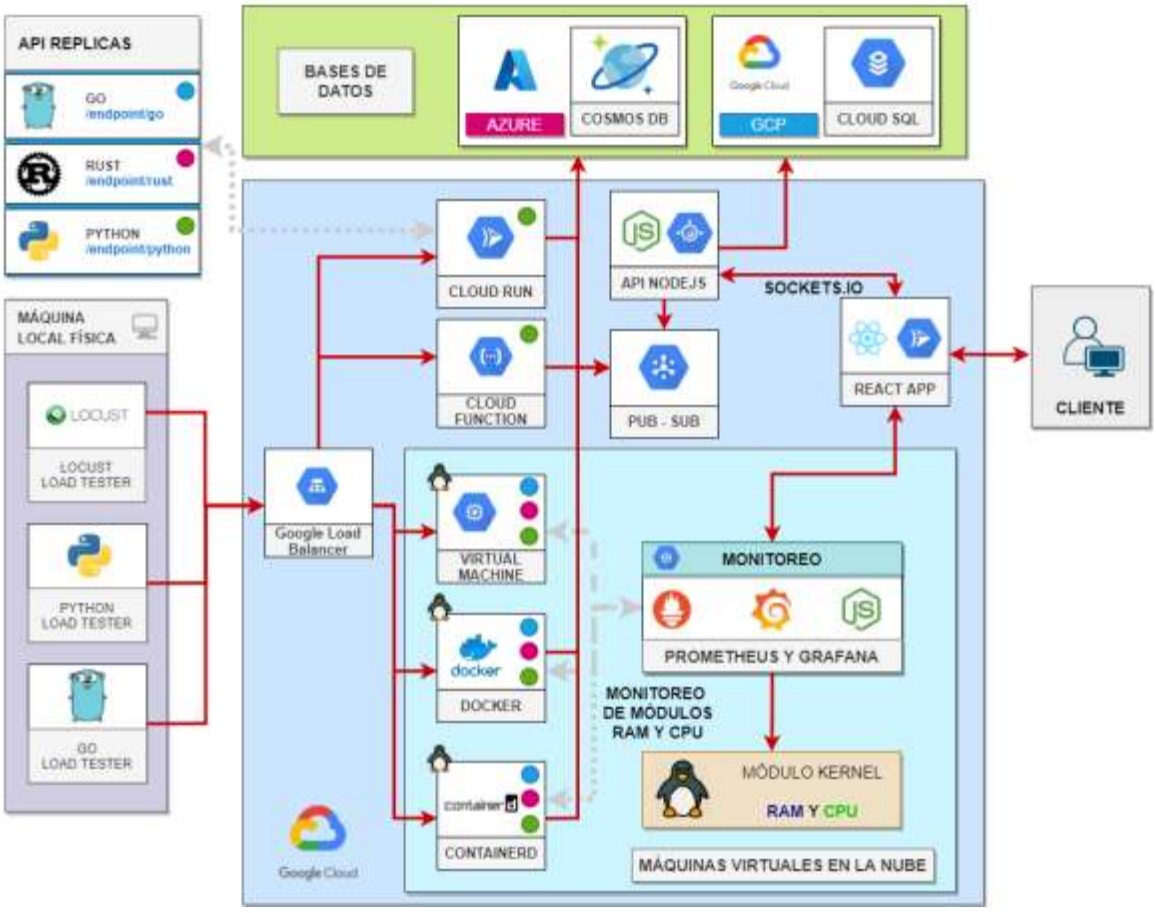
| Nombre | Carnet |
|---|---|
| **Edi Yovani Tomas Reynoso** | 201503783 |
| **Luis Alfonso Ordo;ez Carrillo** | 201603127 |
| **Helmut Efraín Najarro Álvarez** | 201712350 |

# INTRODUCCION

Es un sistema totalmente en la nube, este proyecto consiste en utilizar servicios de Google Cloud Plataform y una base de datos en cloud SQL y una no SQL en Cosmo DB en Microsoft Azure. Con lo que se contara con una carga masiva de datos a partir de los generadores de tráfico, el cual se mostrara de forma detalla en los reportes gráficos etc.

La aplicación consistes en enviar varios tweets y Notificaciones para que sean publicados en una página web además se mostrar la información de la Memoria RAM y sus procesos y el comportamientos de las máquinas virtuales.

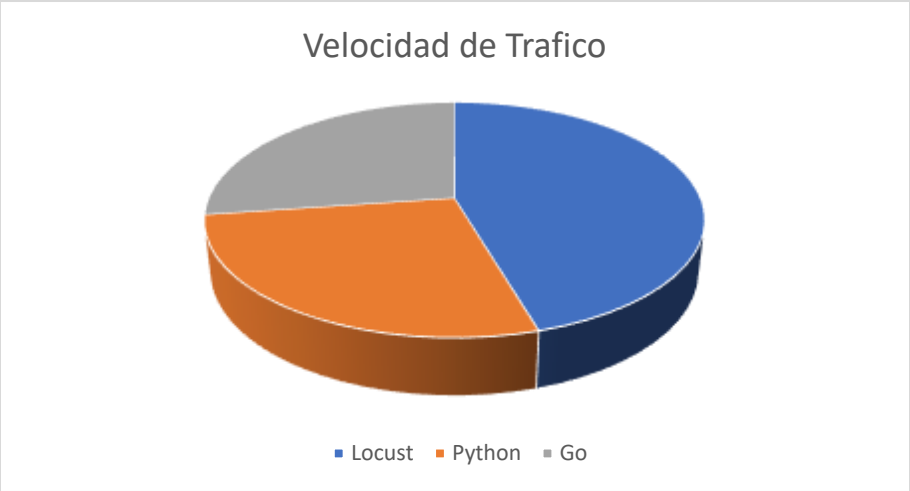
## Diagrama General



## Generadores de tráfico (Fundamentado con gráficas y pruebas)

### ¿Qué generador de tráfico es más rápido?

El generador más rápido para esta ocasión es locust, porque te permite por medio de una interfaz gráfica mandar una cierta cantidad de usuarios a través del tiempo.




### ¿Qué diferencias hay entre las implementaciones de los generadores de tráfico?

Las diferencias entre los generadores de tráfico son de algunos como locust tiene interfaz y puede mandar demasiados datos con cierta cantidad de usuarios al mismo tiempo en cambio los demás o generadores de trafico se tiene que hacer por consola, insertando una ruta quemada o por defecto

**Locust:** En locust se realiza el tráfico más fácilmente y además lo hace con diferentes usuarios finales.

**Golang:** En go se utilizó concurrencia para aprovechar el tiempo entre peticiones.

**Python:** En Python se utilizó flask es una librería la cual se encarga de realizar los endpoints y el api para los generadores de tráfico.

.

## Lenguaje óptimo (Fundamentado con gráficas y pruebas)

¿Qué lenguaje de programación utilizado para las Apis fue más óptimo con relación al tiempo de respuesta entre peticiones?

Golang es lenguaje más optimo entre velocidad de peticiones más que Python, Python tiene mucho más soporte y Rust. es un lenguajes que es menos optimo.

¿Qué lenguaje tuvo el performance óptimo?

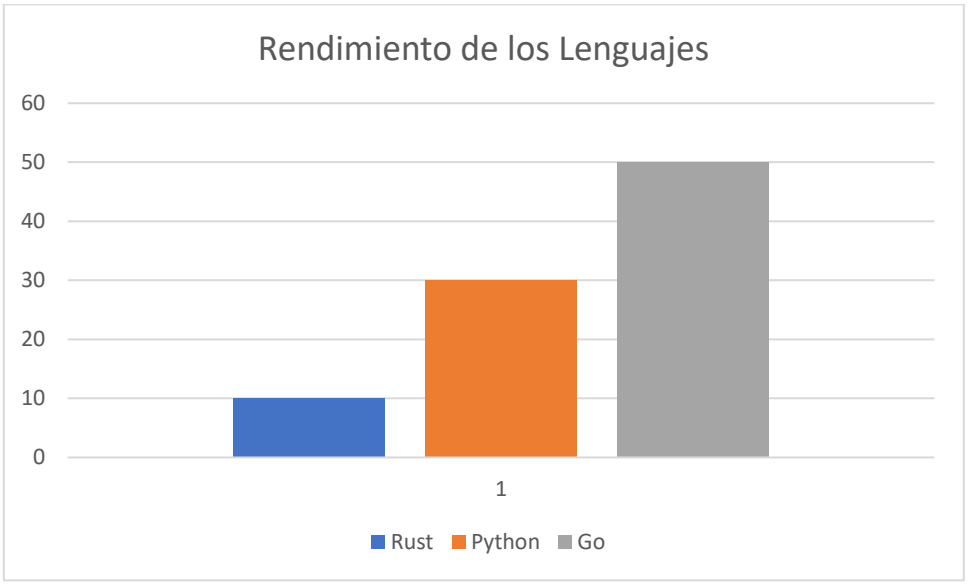
Para las necesidades de su servidor backend, Python ha demostrado ser "lo suficientemente rápido" para la mayoría de las aplicaciones, aunque si necesita más rendimiento, Go lo tiene. Se oxida aún más, pero se paga con el tiempo de desarrollo. Go no está muy lejos de Python en este sentido, aunque ciertamente es más lento de desarrollar, principalmente debido a su pequeño conjunto de características. Rust tiene muchas funciones, pero administrar la memoria siempre llevará más tiempo que dejar que el idioma lo haga, y esto compensa tener que lidiar con la minimizad de Go.

En primer lugar, vamos a comparar el rendimiento de los lenguajes, y qué mejor manera de hacerlo que simplemente resolviendo funciones matemáticas complejas. Si bien no es del todo justo, sin duda lo lleva a la realidad cuando se habla del uso de la memoria y el tiempo dedicado a resolver el problema.

Resolvimos tres problemas diferentes utilizando el lenguaje, a saber, la ecuación de Mandelbrot, el problema de n cuerpos y fasta. Estos son problemas realmente complejos que requieren mucho cálculo y sirven como una manera perfecta de probar el rendimiento y la gestión de la memoria del lenguaje en cuestión. Aparte de eso, son problemas realmente interesantes y vale la pena leerlos, pero por ahora, veamos cómo les va a Golang y Python.


**EN MI OPINION** Golang es el lenguaje más optimo y rápido y casi son iguales con Python en sentido de rendimiento en cambio Rust es un lenguaje que es lento y además complejo

## Gráfico de Rendimientos



## Servicios de GCP

¿Cuál de los servicios de Google Cloud Platform fue de mejor para la implementación de las APIs?

Cloud Run

¿Cuál fue el peor?

Cloud Functions

¿Por qué?

Impráctico pésimo, muy lento, esperar demasiado para los cambios

## Containerd o Docker

¿Considera que es mejor utilizar Containerd o Docker y por qué?

Para las necesidades ninguno es mejor que otro pero en realidad las desventajas es de que Docker tiene mucha documentación detallas a diferencia de Containerd que apenas tiene una pagina y que tiempo poca documentación en este caso para nosotros es Docker, porque tiene muchas más librerías y es mucho más fácil de aprender que containerd.

containerd es un tiempo de ejecución de contenedor de alto nivel que proviene de Docker e implementa la especificación CRI.

Tenemos que empezar con Docker porque es la herramienta de desarrollo más popular para trabajar con contenedores. Y para mucha gente, el nombre "Docker" en sí mismo es sinónimo de la palabra "contenedor".

## Mejor base de datos (Fundamentado con gráficas y pruebas)

¿Qué base de datos tuvo la menor latencia entre respuestas y soportó más carga en un determinado momento?

La base de datos con menos latencia es mongodb en cosmodb porque se tarda en cargar datos

¿Cuál de las dos recomendaría para un proyecto de esta índole?

Se recomiendo usar Mysql es una base de datos demasiado rápido para cargar datos


## Prometheus & Grafana

Considera de utilidad la utilización de Prometheus y Grafana para crear dashboards,

si

¿Por qué?

Si es de utilidad utilizar este software porque tiene librerías interesantes para poder desarrollar.

## Como se crean la Apis en cada lenguaje

En esta ocasión se observa como se crea las Apis en la siguiente imagen

### Python

```python
@app.route('/iniciarCarga/python', methods=['GET'])
def iniciarCarga():
    global cargar, contadorSQL, contadorCosmos, idCosmos

    if cargar == False:
        cargar = True
        contadorSQL = 0
        contadorCosmos = 0
        return jsonify({'message': 'Se ha iniciado la conexion correctamente, puede enviar datos'})

    return jsonify({'message': 'Ya hay una conexion iniciada'})
```

Iniciar carga  es un método get y sirve para iniciar carga

```python
@app.route('/publicar/python', methods=['POST'])
def publicar():
    global cargar
    if not cargar:
        return jsonify({'message': 'Debe iniciar una conexion para poder ingresar datos a la DB'})
    body = request.get_json()

    publicarSQL(body)
    publicarCosmos(body)
    return jsonify({'message': 'Se ha cargado el dato exitosamente a la DB'})
```

Publicar este método sirve para inserta los datos a la base de datos.y es un método post

### Rust

```rust
#[tokio::main]
pub async fn main() {

    dotenv().ok();
    let app = Router::new()
            .route("/iniciarcarga/rust/", get(iniciar_cargar))
            .route("/publicar/rust/", post(post_publicar_carga))
            .route("/finalizarcarga/rust/", post(finalizar_carga));

    // run our app with hyper
    // `axum::Server` is a re-export of `hyper::Server`
    let addr = SocketAddr::from(([0, 0, 0, 0], 4000));
    println!("listening on {}", addr);
    axum::Server::bind(&addr)
    .serve(app.into_make_service())
    .await
    .unwrap();

}
```

Para Rust las Apis se crean de esta manera utilizando un método llamado router adentro se coloca los endpoints correspondientes y luego se le coloca las palabras reservas que son get y pos y adentro se le coloca el nombre del métodos a la cual va hacer los endpoint.

En el siguiente código es donde se va llamar el servidor en puerto 4000 para levantar las apis.

```rust
pub async fn iniciar_cargar()->Json<Mensaje>{
    unsafe {
      if CARGAR == false{
            CARGAR = true;
            CONTADORCOSMODB = 0;
            CONTADORSQLDB = 0;
            let smsjson = Mensaje { mensaje: "Se ha realizado la conexion exitosamente".to_string() };
            return Json(smsjson);
      };
    }
    let smsjson = Mensaje { mensaje: "Actualmente estas Conectado!".to_string() };
    Json(smsjson)
}
```

Este es método cargar “iniciar carga” el cual se encarga de iniciar la carga y es un método “get”.

```rust
pub async fn post_publicar_carga(Json(_req): Json<Tuits>)-> impl IntoResponse {

     let now = Instant::now();
     thread::sleep(Duration::new(1, 0));
     let database_url = env::var("DATABASE_URL").expect("DATABASE URL is not in .env file");
     let client_options = ClientOptions::parse(&database_url).await.unwrap();
     let client = Client::with_options(client_options).unwrap();
     let db = client.database("Olympics");
     let collection = db.collection::<Tuits>("Tuits");
     unsafe{
                       let  _tuiteo = Tuits {
                            nombre: _req.nombre.to_string(),
                            comentario: _req.comentario.to_string(),
                            fecha: _req.fecha.to_string(),
                            hashtags:_req.hashtags.to_vec(),
                            upvotes: _req.upvotes,
                            downvotes: _req.downvotes
                          };
          collection.insert_one(_tuiteo, None).await.unwrap();
          CONTADORCOSMODB += 1;
     }
```

Se muestra donde el método “post” y sirve para  insertar datos.

## Go

Método startload a se encarga de iniciar la conexión a la base de datos

```
func startLoad(c *gin.Context) {

    if !ready {

        _, err := cos.Connect()
        if err != nil {
            ready = false
            c.JSON(http.StatusInternalServerError, "Cosmos DB failed :(")
        }

        err1 := sql.Init()
        if err1 != nil {
            ready = false
            c.JSON(http.StatusInternalServerError, "SQL CLoud failed :(")
        }else{
            ready = true
            c.JSON(http.StatusInternalServerError, "All set!")
        }

    }else{
        c.JSON(http.StatusInternalServerError, "Connection already started")
    }

}
```

Este método se encarga de publicar los tuits en la base de datos

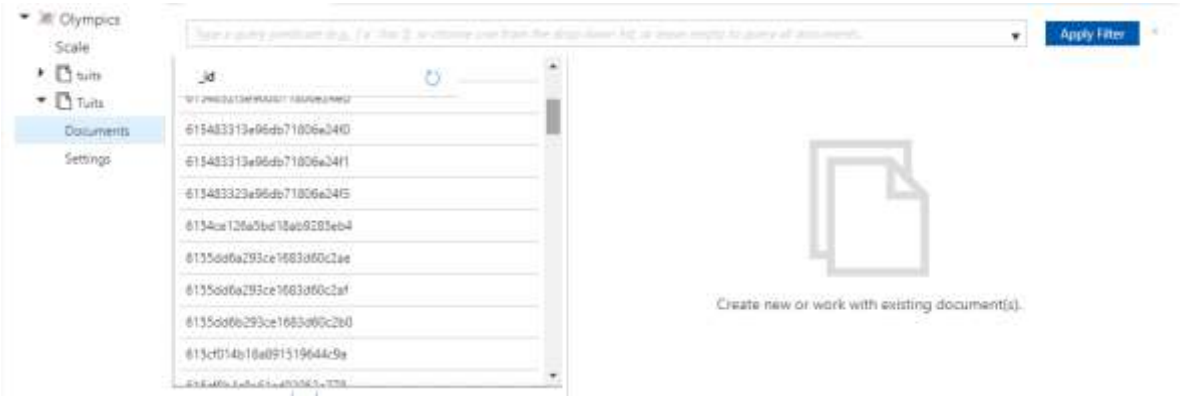
```
func postTuitCosmos(c *gin.Context) {
    t := time.Now()
    var newTuit ts.Tuit

    fmt.Println("====================== POSTING TUIT IN COSMOS ======================")

    // Call BindJSON to bind the received JSON to
    if err := c.BindJSON(&newTuit); err != nil {
        return
    }

    tuits = append(tuits, newTuit)
    msg, err := cos.Create(newTuit)

    if err!=nil{
        fmt.Println(err)
        cosmosLogs = append(cosmosLogs, ts.Log{ StatusNumber:http.StatusInternalServerError, Message:fmt.Sprint(err), Time:time.Since(t) })
        c.JSON(http.StatusInternalServerError, ts.Log{StatusNumber:http.StatusInternalServerError, Message:fmt.Sprint(err), Time:time.Since(t) })
    }else{
        cosmosLogs = append(cosmosLogs, ts.Log{ StatusNumber:http.StatusCreated, Message:msg, Time:time.Since(t) } )
        c.JSON(http.StatusCreated, ts.Log{ StatusNumber:http.StatusCreated, Message:msg, Time:time.Since(t) })
    }

}
```

Se mostro algunas partes del código donde se realizar las Api donde se hace el tráfico de datos para poder insertar a la base de datos sql y nosql para poder mostrar toda esa información en la parte del cliente y además se realiza los reportes para poder tener un mejor detalle de los flujos.

También se muestra la única tabla de entidad de relación que se utilizo


En la parte de abajo es donde se muestra el código de la única tabla que se utilizo en la base de datos .

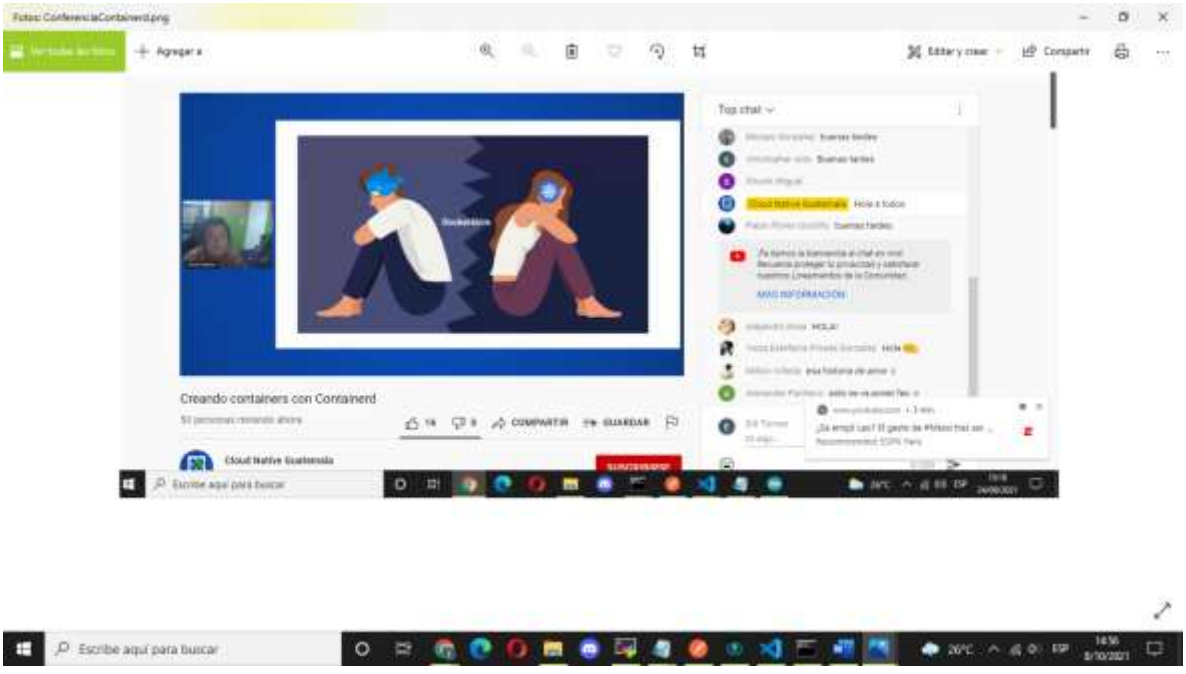
```
CREATE TABLE OLIMPIC(
    idOlimpic INT identity(1,1) NOT NULL,
    nombre CHAR(50),
    comentario CHAR(300),
    fecha DATE,
    hashtags VARCHAR(300),
    upvotes INT,
    downvotes INT,
    PRIMARY KEY(idOlimpic)
);
```



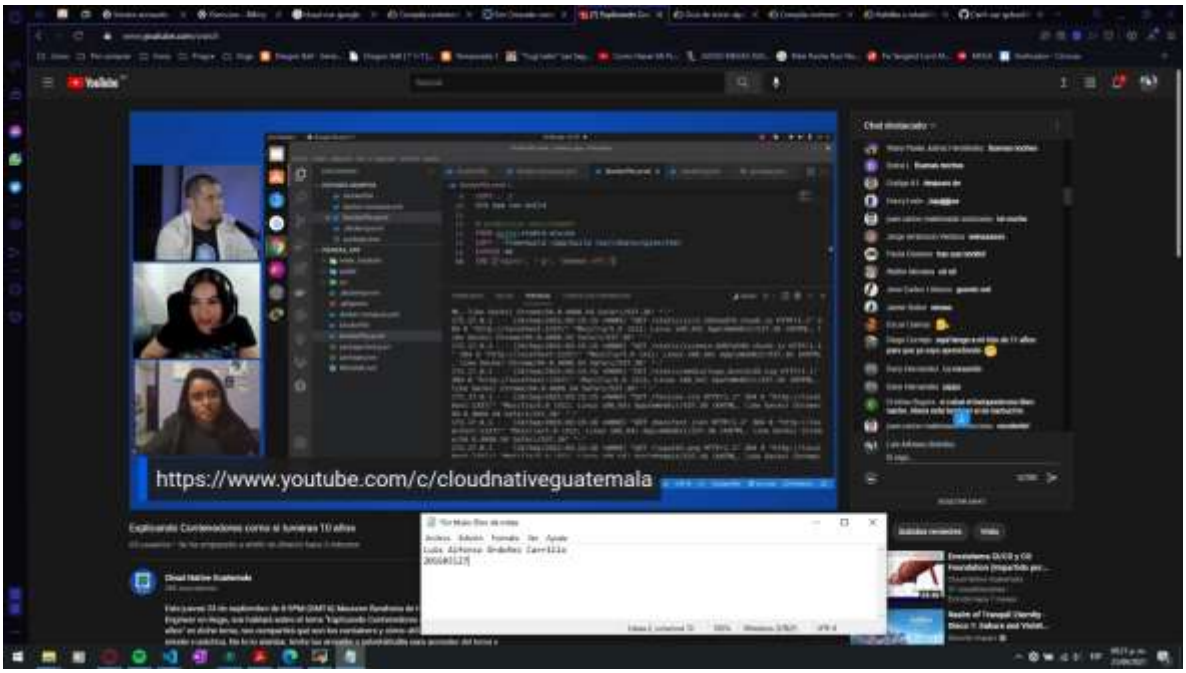

Aquí se muestra el registro de la base de datos no sql  se muestra algunas colecciones  y la forma de inserción es por medio de formatos Json.

Conferencias

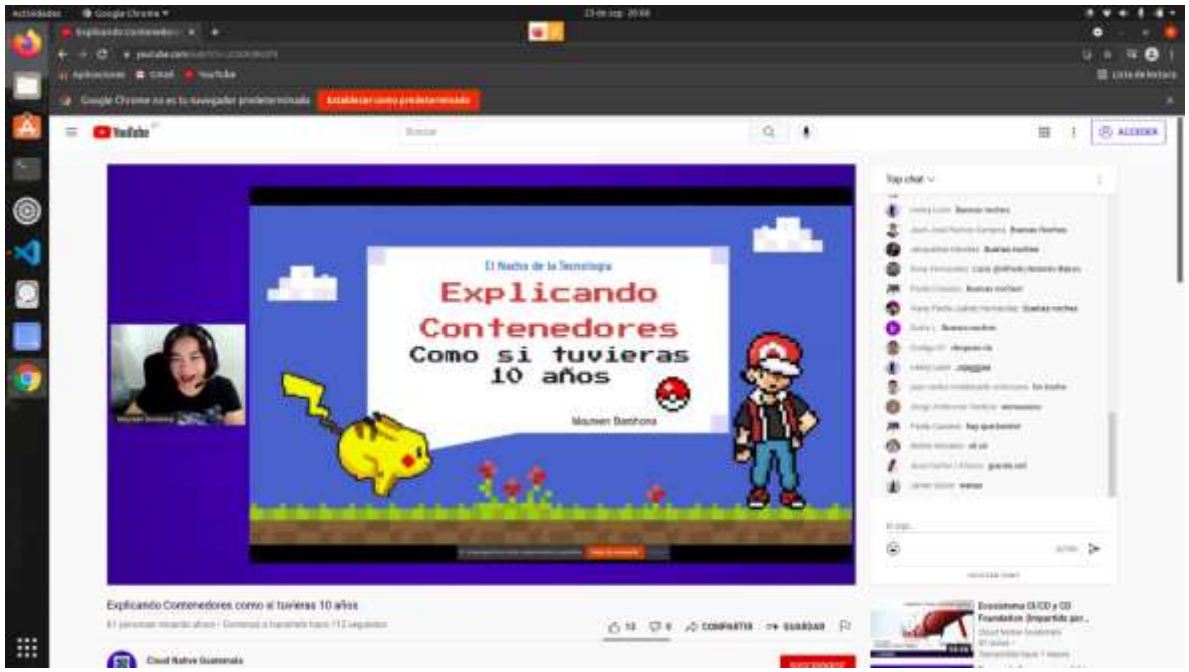
Edi Yovani Tomas Reynoso

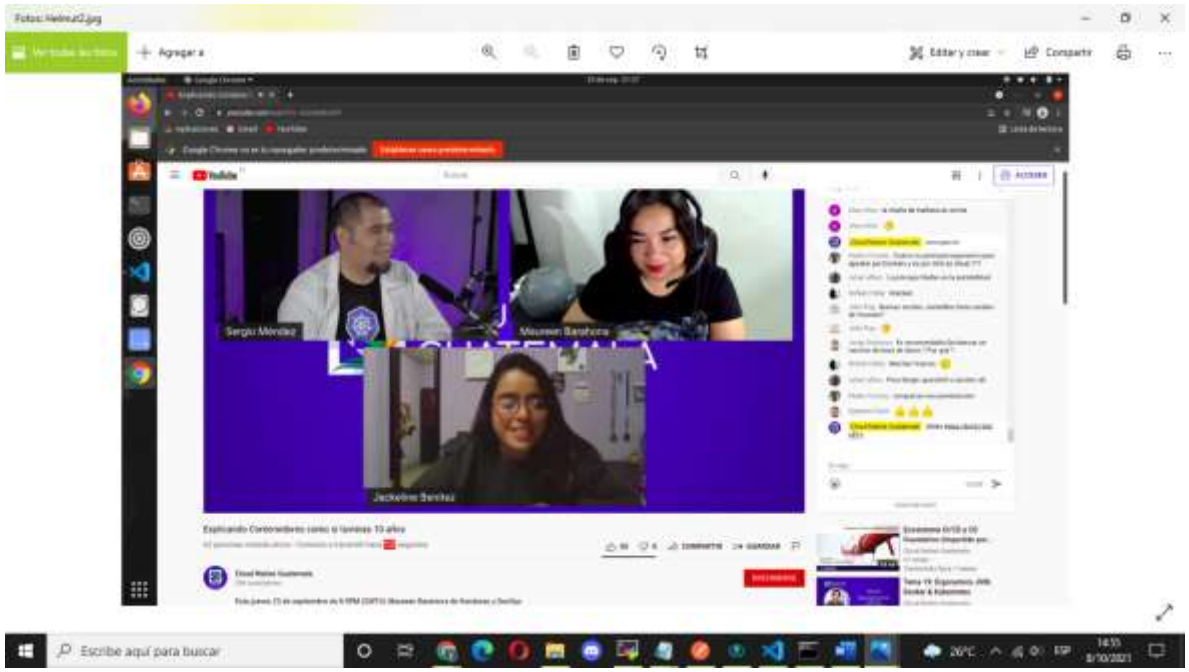




Luiz Ramirez




Helmut


El Nacho de la Tecnología
Explicando
Contenedores
Como si tuvieras
10 años
Maureen Barahona
Explicando Contenedores como si tuvieras 10 años


Sergio Méndez
Maureen Barahona
Jackeline Benitez
Explicando Contenedores como si tuvieras 10 años
Escribe aquí para buscar